Windows Phone 7

IN ACTION

Timothy Binkley-Jones
Massimo Perga
Michael Sync

MANNING

Windows Phone 7 in Action

Windows Phone 7 in Action

TIMOTHY BINKLEY-JONES
MASSIMO PERGA
MICHAEL SYNC

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2013 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Jeff Bleiel
Copyeditor: Benjamin Berg
Technical proofreader: Richard Reukema
Proofreader: Melody Dolab
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617290091
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 18 17 16 15 14 13 12

brief contents

PART 1 INTRODUCING WINDOWS PHONE 1

1 ■ A new phone, a new operating system 3
2 ■ Creating your first Windows Phone application 29

PART 2 CORE WINDOWS PHONE 55

3 ■ Fast application switching and scheduled actions 57
4 ■ Launching tasks and choosers 93
5 ■ Storing data 121
6 ■ Working with the camera 149
7 ■ Integrating with the Pictures and Music + Videos Hubs 171
8 ■ Using sensors 199
9 ■ Network communication with push notifications and sockets 227

PART 3 SILVERLIGHT FOR WINDOWS PHONE 257

10 ■ ApplicationBar, Panorama, and Pivot controls 259
11 ■ Building Windows Phone UI with Silverlight controls 284

12 ■ Manipulating and creating media with MediaElement 310

13 ■ Using Bing Maps and the browser 341

PART 4 SILVERLIGHT AND THE XNA FRAMEWORK 369

14 ■ Integrating Silverlight with XNA 371

15 ■ XNA input handling 399

contents

preface xv
acknowledgments xvi
about this book xvii
about the cover illustration xxi

PART 1 INTRODUCING WINDOWS PHONE 1

1 *A new phone, a new operating system 3*
1.1 Rebooting the Windows Phone platform 4
1.2 Windows Phone foundations 5
Hardware specs 6 ▪ A new user interface 7
User experience 8 ▪ Platform APIs and frameworks 10
AppHub and the Windows Phone Marketplace 11
1.3 Comparing Windows Phone to other mobile platforms 12
Windows Mobile 12 ▪ Apple iOS 14 ▪ Android 17
1.4 The Windows Phone Developer Tools 20
Visual Studio for Windows Phone 20 ▪ Expression Blend for Windows Phone 20 ▪ XNA Game Studio 20
Windows Phone Emulator 21 ▪ Windows Phone Developer Registration tool 22 ▪ XAP Deployment tool 23

WPConnect 24 ▪ Isolated Storage Explorer tool 25
Marketplace Test Kit 25

1.5 Summary 28

2 Creating your first Windows Phone application 29

2.1 Generating the project 30

Debugging phone projects 33 ▪ Application startup 34

2.2 Implementing Hello World 35

Customizing the startup page 35 ▪ Adding application content 37 ▪ Adding the greetings page 39

2.3 Interacting with the user 41

Touch typing 41 ▪ Touch gestures 42
Adding a toolbar button 43

2.4 Page navigation 45

Navigating to another page 45 ▪ Passing parameters between pages 47 ▪ Changing the Back key behavior 48
Navigating with tiles 49

2.5 Application artwork 50

Customizing the splash screen 50
Customizing tile images and application icons 50

2.6 Try before you buy 52

2.7 Summary 53

PART 2 CORE WINDOWS PHONE 55

3 Fast application switching and scheduled actions 57

3.1 Fast application switching 58

Understanding lifetime events 59
Creating the Lifetime sample application 61

3.2 Launching the application 62

Construction 62 ▪ First-time initialization 65

3.3 Switching applications 66

Going dormant 66 ▪ Returning to action 68
Tombstoning 69

3.4 Out of sight 74

Obscuration 74 ▪ Running behind the lock screen 75

3.5 Working on a schedule 77

Introducing the Scheduled Action Service 78
Scheduling a reminder 81 ▪ Editing a notification 83
Deleting a notification 84

3.6 Creating a background agent 85

Background agent projects 85 ▪ Executing work from the background agent 86 ▪ Scheduling a PeriodicTask 87
Scheduled tasks expire after two weeks 88
User-disabled tasks 89 ▪ When things go awry 90
Testing background agents 91

3.7 Summary 92

4 *Launching tasks and choosers 93*

4.1 Tasks API 94

4.2 Launchers 96

Placing a phone call 97 ▪ Writing an email 98
Texting with SMS 99 ▪ Working with the Marketplace 100
Searching with Bing 103

4.3 Choosers 103

Completed events 104 ▪ Saving a phone number 105
Saving an email address 106 ▪ Saving a ringtone 107
Choosing a phone number 108 ▪ Choosing an email address 109 ▪ Choosing a street address 109

4.4 UserData APIs 110

Searching for contacts 111 ▪ Reviewing appointments 115

4.5 Summary 119

5 *Storing data 121*

5.1 Creating the High Scores sample application 122

Displaying the high score list 123 ▪ Managing the high score list 125 ▪ Defining a high score repository 126

5.2 Storing data with application settings 127

5.3 Serializing data to isolated storage files 129

Serializing high scores with the XmlSerializer 130
Deleting files and folders 131

5.4 Working with a database 132

Attributing your domain model 133 ▪ Defining the data context 135 ▪ Creating the database 136

CRUD operations 137 ▪ Searching for data 141
Compiling queries 142 ▪ Upgrading 143
Adding a read-only database to your project 146

5.5 Summary 147

6 *Working with the camera 149*

6.1 Starting the PhotoEditor project 150

6.2 Working with the camera tasks 151

Choosing a photo with PhotoChooserTask 151
Taking photos with CameraCaptureTask 154
Handling picture orientation in CameraCaptureTask 155

6.3 Controlling the camera 159

Painting with the VideoBrush 162 ▪ Snapping a photo 163
Supporting fast application switching 165

6.4 Image editing 165

Rendering Silverlight elements 166 ▪ Saving an image to isolated storage 167 ▪ Loading an image from isolated storage 168

6.5 Summary 169

7 *Integrating with the Pictures and Music + Videos Hubs 171*

7.1 Working with pictures in the Media Library 172

Exposing Pictures 172 ▪ Saving pictures to the media library 174 ▪ Retrieving a picture from the media library 175

7.2 Editing and sharing from the Pictures Hub 176

Extending the Picture Hub 176 ▪ Extending the Picture Viewer 178 ▪ Sharing pictures from your Pictures Hub extension 180

7.3 Playing and recording with the Music + Videos Hub 181

Enabling XNA Framework events 183 ▪ Building the user interface 183 ▪ Recording audio 185
Playing audio 189

7.4 Playing recorded audio in the Music + Videos Hub 190

Fulfilling Music + Videos Hub requirements 191
Launching from the Music + Videos Hub 193

7.5 Playing recorded audio with a background agent 194

7.6 Listening to FM radio 196
7.7 Summary 197

8 *Using sensors 199*

8.1 Understanding the sensor APIs 200
Data in three dimensions 201 ▪ Reading data with events 202 Polling for data 203
8.2 Creating the sample application 203
Creating a reusable Bar control 204 ▪ Designing the main page 206 ▪ Polling sensor data with a timer 207
8.3 Measuring acceleration with the accelerometer 208
Hooking up the sensor 209 ▪ Acceleration in the emulator 210 Interpreting the numbers 211
8.4 Finding direction with the Compass 213
Hooking up the sensor 214 ▪ Interpreting the numbers 216 Calibrating the sensor 217
8.5 Pivoting with the Gyroscope 217
Hooking up the sensor 218
8.6 Wrapping up with the motion sensor 219
Building a motion enabled sample application 220 Hooking up the sensor 222 ▪ Interpreting the numbers 224
8.7 Summary 226

9 *Network communication with push notifications and sockets 227*

9.1 Detecting network connectivity 228
Reading device settings 229 Using the NetworkInterface class 231
9.2 Pushing notifications to a phone 232
Three types of notifications 233 ▪ Push notification workflow 234 ▪ Creating a Push Notification client 235 Opening a notification channel 236 ▪ Looking for navigation parameters 237 ▪ In-app notifications 238 Copying the channel URI 239
9.3 Simulating a push notification service 239
Issuing HTTP web requests 240 ▪ Sending toast notifications 243 ▪ Using notifications to update a tile 244
9.4 Tiles without all the pushiness 247

9.5 Communicating with sockets 249
9.6 Implementing a chat application with TCP sockets 250
Building the Chit-chat client 250 ▪ Connecting to the server 252
Receiving messages from the server 254 ▪ Sending a message 255
9.7 Summary 256

PART 3 SILVERLIGHT FOR WINDOWS PHONE 257

10 *ApplicationBar, Panorama, and Pivot controls 259*
10.1 Working with the ApplicationBar 260
Building an application bar 261 ▪ Tooling support 262
Changing the application bar appearance 264
Dynamically updating buttons and menu items 265
Designing button icons 266
10.2 Improving the scenery with the Panorama control 268
Building a panorama application 269 ▪ Widen up the view 271
Remembering where you are 272 ▪ Adding a background 273
Customize the title 274
10.3 Pivoting around an application 275
Building the sample 276 ▪ Remembering the current selection 278 ▪ Generating sample data 279
Dynamically loading pages 281
10.4 Summary 283

11 *Building Windows Phone UI with Silverlight controls 284*
11.1 Handling page orientation 285
Supported orientations 286 ▪ Animating orientation transitions 287 ▪ Changing orientation 289
11.2 Building user interfaces 290
TextBlock 290 ▪ Border 292 ▪ Shapes 293
ProgressBar 293 ▪ Image 294
11.3 Receiving Input 295
Button 295 ▪ HyperlinkButton 296 ▪ CheckBox 297
RadioButton 297 ▪ TextBox 298 ▪ Slider 300
11.4 Silverlight Toolkit for Windows Phone 301
ToggleSwitch 302 ▪ DatePicker and TimePicker 303
ContextMenu 304 ▪ GestureListener 306
11.5 Summary 308

12 *Manipulating and creating media with MediaElement* 310

12.1 Building a media player with MediaElement 312

Creating the media player project 312 ▪ Loading media files 315 Media element states 317 ▪ Controlling volume 319

12.2 Manipulating the media stream with MediaStreamSource 320

Opening a media source 321 ▪ Seeking media 322 Sampling media 323

12.3 Creating custom video 324

Initializing the stream source 325 ▪ Opening the video stream source 326 ▪ Generating media samples 327

12.4 Creating custom audio 329

Defining a custom audio stream source 330 Opening the audio stream source 331 Generating audio samples 332

12.5 Streaming media clients 334

Using Smooth Streaming 335 ▪ Streaming limitations 338

12.6 Summary 340

13 *Using Bing Maps and the browser* 341

13.1 Introducing Bing Maps 342

Preparing the application 343 ▪ Launching the Bing Maps application 343 ▪ Finding directions 344

13.2 Location services 346

Building the sample application 346 Hooking up the service 348

13.3 Embedding a Map control 352

Mapping the current location with the GeoCoordinateWatcher 353 Creating a push pin 354

13.4 Using the Bing Maps Services 355

Adding the service reference 355 ▪ Reverse geocoding 356

13.5 Building an HTML 5-based application 358

Launching Internet Explorer 359 ▪ Embedding Internet Explorer 360 ▪ Adding HTML pages to the project 361 Matching the Metro style 363 ▪ Working from Isolated Storage 364 ▪ Bridging C# and JavaScript 366

13.6 Summary 368

PART 4 SILVERLIGHT AND THE XNA FRAMEWORK 369

14 *Integrating Silverlight with XNA* 371
14.1 Creating a Silverlight with XNA application 373
Sharing the graphics device 374 ▪ The game loop 375
14.2 Building the game page 376
Understanding models 377 ▪ Rendering models 379
Adding shapes 382 ▪ Moving around 383
Running a demonstration 387 ▪ Don't repeat yourself 389
Collecting shapes 390 ▪ It's the end of the world 393
14.3 Implementing a scoreboard with Silverlight 394
Adding a scoreboard 395 ▪ Rendering the texture 396
14.4 Summary 397

15 *XNA input handling* 399
15.1 Implementing pause and resume 401
Pausing game play 401 ▪ Adding the resume button 402
15.2 Adding input services 404
Choosing an input type 404 ▪ Creating a thumbstick 407
Creating a button pad 411 ▪ Gaming with gestures 415
Moving with the motion sensor 418
15.3 Summary 421

appendix A Microsoft Expression Blend for Windows Phone 423
appendix B Silverlight and the Extensible Application Markup Language 430
appendix C AppHub and Marketplace 438

index 445

preface

We've come from different backgrounds and locations to write this book—Michael is a Silverlight MVP who lives in Singapore; Massimo lives in Europe and worked at Microsoft on the Windows Phone team; and Timothy lives in the United States and was the technical proofreader for other Manning books on WPF and Silverlight. Against all this diversity, our shared passion for Silverlight and mobile applications brought us together to produce this book.

In 2011, nearly half a billion smart phones were sold worldwide. The world is quickly moving to a fully connected society, and smart phones like the Windows Phone are positioned to play a major role in how we access data, connect with our family and friends, and generally interact with the world around us. Smartphones are almost always with us, know where they are located, and are connected to the internet.

Our job as application developers is to create applications that can interact with our environment, sift through the data, and present a simplified view of the world to users overwhelmed with the complexities of the fast-paced, high-tech, digital world. We hope our book gives you the knowledge you need to determine location, process sensor input, capture audio and video, and scrutinize data to build killer Windows Phone applications that integrate nicely with the operating system and native applications.

acknowledgments

We would like to thank our family, friends, and coworkers for their support and advice, for being there when we needed someone to listen to half-formed ideas, and for understanding when we said "I'd love to, but I have to work on the book." The chapters covering the XNA Framework would have been impossible to write without advice and debugging from Trystan Binkley-Jones.

Of course none of this would have been possible without Microsoft and the support they provide to the development community. In particular, we would like to thank Cliff Simpkins for providing hub screenshots and a developer phone complete with a pre-release version of Windows Phone 7.5.

We would like to thank the following reviewers, who read the manuscript at various stages during development and provided invaluable feedback: 'Anil' Radhakrishna, Berndt Hamboeck, Dave Campbell, Francesco Goggi, Jedidja Bourgeois, Lester Lobo, Loïc Simon, Mark Monster, Nishant Sivakumar, Scott Turner, Steve Grey-Wilson, and Vipul Patel. Special thanks to Richard Reukema for his careful technical proofread of the manuscript shortly before it went into production.

Finally, our thanks to everyone at Manning, especially Marjan Bace, Michael Stephens, and our development editor Jeff Bleiel, as well as our production team of Benjamin Berg, Melody Dolab, Dennis Dalinnik, Janet Vail, and Mary Piergies. Your guidance and support during the writing and production process were much appreciated.

about this book

This book is a hands-on guide to building mobile applications for Windows Phone 7.5 using Silverlight, C#, XNA, or HTML5. The Windows Phone 7 operating system is Microsoft's latest entry into the fiercely competitive mobile market. Windows Phone 7 is not an upgrade of previous mobile operating systems, Windows Mobile and Windows Phone 6.5. Microsoft has reimagined what a mobile operating system should be and completely changed the rules on how to build mobile applications.

To power the phone, Microsoft started with familiar foundations in Windows CE the .NET Compact Framework, and the Zune user interface, adapted the Silverlight and XNA libraries, and then added entirely new APIs for interacting with mobile hardware, sensors, and software. In this book we show you how to build user interfaces that adhere to the new Metro design, and how to use the new APIs to access the sensors and integrate with the built-in application.

Who should read this book

This book is written for C# and .NET developers who are familiar with XAML, Silverlight or WPF development. This book does not teach you the subtleties of C# or Silverlight/XAML development. That being said, the book avoids many of the more powerful features of Silverlight and the Model-View-ViewModel pattern used by many Silverlight developers. Instead we kept the focus on the features and APIs that are unique to the phone and endeavored to make the content accessible to those readers who are not very familiar with Microsoft technologies.

Roadmap

This book has four parts, fifteen chapters, and three appendices. We divided the book into sections that introduce Windows Phone 7, cover the core concepts of the phone, and discuss enhancements to Silverlight. The final section of the book shows you how Silverlight applications can use the powerful graphics API found in the XNA Framework.

Part 1 is an introduction to Windows Phone, the developments, and the SDK. This part walks you through creating your first application.

In chapter 1, you'll discover why Microsoft scrapped the Windows Mobile operating system in favor of a completely new smartphone platform. We compare Windows Phone 7 to Android and iOS development and introduce you to Visual Studio and the SDK tools you'll use when building applications.

In chapter 2 you'll build your first Windows Phone 7 project which is a traditional Hello World application. We use the Hello World application to introduce you to touch events, application tiles, the application bar, and the Windows Phone navigation model.

Part 2 examines the core Windows Phone platform and what makes developing for the phone different from developing for the desktop or the browser. We'll introduce concepts that are brand new to Windows Phone, as well as concepts that have been adapted to operate within the phone's limitations.

In chapter 3 you'll learn about Fast Application Switching, Microsoft's name for the battery-saving technology that allows a dormant application to be quickly restored when a user switches from a foreground application to a background application. You'll also learn how to create background agents that run periodically.

In chapter 4 you'll read about how to use Launchers and Choosers to interact with built-in applications such as the phone dialer, email, and the People Hub.

In chapter 5 you'll store application data using isolated storage and a SQL database.

In chapter 6 you'll build an application that captures images from the phone camera and allows a user to make simple modifications to the photographs.

In chapter 7 you'll integrate an application with the built-in Pictures and Music + Video Hubs.

In chapter 8 you'll learn how to obtain data from the phone's hardware including the accelerometer, compass, gyroscope, and motion sensor.

In chapter 9 we cover networking topics such as using TCP sockets and Push Notifications. Push Notifications provide the ability for an external application or web service to send messages and updates to a particular Windows Phone device.

Part 3 presents new Silverlight features and controls used to build applications that match the look and feel of Windows Phone.

In chapter 10 you'll take a deep dive into the Application Bar, Panorama, and Pivot controls that are unique to the Windows Phone.

In chapter 11 you'll learn how to build applications that automatically adjust themselves to match the Metro design, and how to control the software keyboard. You will also be introduced to the Silverlight Toolkit for Windows Phone, a Codeplex project from Microsoft.

In chapter 12 you'll work with the MediaElement to play audio and video and will learn how to create a Windows Phone 7 Smooth Streaming client application.

In chapter 13 you'll build a location-aware application using location services and the Bing Maps API. You'll also build an HTML5-based application.

Part 4 of this book demonstrates how Silverlight and the XNA Framework can be used together to build exciting games and applications. The XNA Framework includes a rich library for three-dimensional modeling and rendering.

In chapter 14 you'll use the Windows Phone Silverlight and XNA Application template to create a Hello World game, and learn the techniques used to render Silverlight user interface elements with the XNA graphics framework. We give you a crash course in XNA concepts such as 3D animation techniques, collision detection, and the game loop.

In chapter 15 you'll continue working with the sample game and learn how to use sprites for 2D graphics and animation. You'll use raw touches, gestures, the motion sensor, and the Mouse API to let a game player wander around the game world.

Code conventions and downloads

All source code in the book is in a `fixed-width font like this`, which sets it off from the surrounding text. In many listings, the code is annotated to point out the key concepts, and numbered bullets are used in the text to provide additional information about the code. We've tried to format the code so that it fits within the available page space in the book by adding line breaks and using indentation carefully. Sometimes, however, very long lines include line continuation markers.

The source code presented in the book can be downloaded from the publisher's web site at www.manning.com/WindowsPhone7inAction.

The source code is organized into folders for each chapter, with sub-folders for each project. The source code contains the completed sample projects for each chapter. Many of the sample projects link to image files shipped as part of the SDK. We chose not to redistribute the image files and instead used Visual Studio's linked file features when adding the images to the projects.

Software or hardware requirements

The Windows Phone Developer Tools, which Microsoft provides as a free download, are required to compile and execute the sample projects presented in this book. The Windows Phone Developer Tools install an express edition of Visual Studio 2010 configured with the phone development tools. If you already have a retail edition of Visual Studio 2010 installed on your computer, the phone development tools will be installed as a plug-in to the IDE. Windows Phone projects can be written in both C# and Visual Basic.

We'll use the express edition throughout the book for the screen shots and sample code. Code and user interface design features will work the same in the retail editions of Visual Studio 2010. You can download the Windows Phone Developer Tools from http://create.msdn.com.

A physical Windows Phone is not required. The Windows Phone Developer Tools include the Windows Phone 7 Emulator. With a few exceptions, the samples in this book will run in the emulator exactly as they would on a physical phone. The samples that integrate with the Music + Videos Hub and the samples that make use of the compass and gyroscope will require a physical device. If you want to use a physical device, a $99 yearly membership to the App Hub is required to unlock your phone.

The Windows Phone 7 Emulator should work on most recent computers. The emulator performs better if your computer has a CPU with virtualization extensions like most of the recent AMD and Intel CPUs. The emulator works best with a DirectX 10 or later graphics card with a WDDM 1.1 driver. The system requirements for the Windows Phone tools are

- Supported operating systems: Windows Vista (x86 and x64) with Service Pack 2—all editions except Starter Edition; Windows 7 (x86 and x64)—all editions except Starter Edition.
- Installation requires 4 GB of free disk space on the system drive.
- 3 GB RAM.
- Windows Phone Emulator requires a DirectX 10 or above capable graphics card with a WDDM 1.1 driver.

Author Online

Purchase of *Windows Phone 7 in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/WindowsPhone7inAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the cover illustration

The figure on the cover of *Windows Phone 7 in Action* is captioned "A soldier." The illustration is taken from a 19th-century edition of Sylvain Maréchal's four-volume compendium of regional dress customs and uniforms published in France. Each illustration is finely drawn and colored by hand. The rich variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade, station in life, or rank in the army was just by their dress.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

Part 1

Introducing Windows Phone

Welcome to *Windows Phone 7 in Action*, where you'll learn all about building applications for Microsoft's newest mobile operating system. This book is divided into four parts; part 1 introduces you to the Windows Phone and the software development kit, and walks you through creating your first application.

In chapter 1 you'll discover why Microsoft scrapped the Windows Mobile operating system in favor of a completely new smartphone platform. We compare Windows Phone 7 to Android and iOS development and introduce you to Visual Studio and the SDK tools you'll use when building applications.

In chapter 2 you'll build your first Windows Phone 7 project, which is a traditional Hello World application. We use the Hello World application to introduce you to touch events, application tiles, the application bar, and the Windows Phone navigation model.

1

A new phone, a new operating system

This chapter covers

- Introducing Windows Phone 7
- Understanding the hardware
- Porting applications from other platforms
- Developing for Windows Phone

Windows Phone 7 is more than a new operating system. Windows Phone 7 is an operating system, a powerful hardware platform, and several web services, all combined to provide a great experience for the busy Life Maximizer. *Life Maximizer* is the term used by Microsoft to represent the target consumers of the new phone. Life Maximizers demand the most from their phones as they balance work and life, and use their phones to manage their busy lifestyles. Windows Phone 7 was designed to let users get tasks done faster and allow them to get back to the important aspects of their life.

The Windows Phone 7 operating system is Microsoft's latest entry into the fiercely competitive mobile market. Windows Phone 7 is not an upgrade of previous mobile operating systems, such as Windows Mobile and Windows Phone 6.5. Microsoft has reimagined what a mobile operating system should be and completely changed the rules on how to build mobile applications. To power the phone, Microsoft started with familiar foundations in Windows CE, the .NET Compact Framework and the Zune user interface, adapted the Silverlight and XNA libraries, and added entirely new APIs for interacting with mobile hardware, sensors, and software. To enable developers, Microsoft created a toolbox composed of Visual Studio, Expression Blend, and XNA Game Studio.

The first version of the Windows Phone 7 operating system was released in October 2010. Microsoft followed the release with an update in the early months of 2011, adding copy/paste support and performance improvements. At the Mix 2011 conference, Microsoft unveiled details about the Windows Phone 7.5 operating system and the corresponding Windows Phone SDK 7.1. The Windows Phone 7.1 SDK includes several new features, such as fast application switching, background agents, access to the camera hardware, and a built-in SQL CE database engine. Windows Phone 7.5 also exposes new compass, gyroscope, and motion sensors.

> **NOTE** We find it a bit confusing that the new operating system is versioned with 7.5 while the corresponding SDK is versioned 7.1. Throughout this book we'll refer to both operating system releases as *Windows Phone 7* or just *Windows Phone*. We'll provide notes and tips when discussing features that are only available in the Windows 7.1 SDK.

In this chapter we present the motivation behind this revolution in the Microsoft OS for mobile devices. We detail how Windows Phone 7 differs from previous mobile operating systems so that you can assess the capabilities of the new platform and understand how existing designs and code can be ported. We describe the minimum hardware specifications common to the different Windows Phone 7 devices so that developers can confidently target equipment that will always be available. Finally, we introduce the developers tools that you'll use throughout the book to build applications and games targeted at the Windows Phone.

1.1 Rebooting the Windows Phone platform

Microsoft has been building operating systems for mobile devices and phones for more than a decade. One of the earliest versions was Pocket PC 2000, running on palm-sized devices like the Hewlett-Packard Jornada and the Compaq iPAQ. These early devices were not smartphones, but were portable computers or PDAs targeted for business users and didn't initially include phone hardware or network connectivity. Users interacted with these devices using a stylus on a single-point touch screen and an awkward hardware-input panel. Pocket PC 2000 was initially built on Windows CE 3.0, and later added the first version of the .NET Compact Framework. Device manufacturers often created custom builds of the operating system tightly coupled to the specific hardware on a single device—making operating system upgrades impossible for most users.

Until Windows Phone 7, the most recent versions of Microsoft's operating system for mobile devices have been Windows Mobile 6 and Windows Phone 6.5. Windows

Mobile 6 is built on Windows CE 5 and includes the .NET Compact Framework 2.0 SP1. Windows Mobile 6 comes in three editions—Standard, Professional, and Classic.

> **NOTE** For the remainder of the book, when the term *Windows Phone* is used without a version number, we are referring to Windows Phone 7.5. We'll use *Windows Mobile* or *Windows Phone 6.5* to refer to older versions of the phone operating system.

Mobile phones have evolved rapidly and incredibly in the past several years. Once intended solely for business users, mobile phones are now predominately consumer devices, and in many cases have replaced land-line services as a user's only phone. Smartphones now include radios, music players, cameras, global positioning systems, compasses, and accelerometers. Single-point touch screens that required a stylus have been replaced with multi-point touch screens that work with your fingertips. Awkward hardware input panels have been replaced with software input panels and optional hardware keypads.

Apple led the smartphone revolution with the release of the iPhone in June of 2007, and the introduction of the App Store in July of 2008. Google followed with the introduction of the Android OS and Market in October of 2008. Since then, Microsoft has seen declines in Windows Mobile's market share as consumers and device manufacturers turn to smartphones running new mobile operating systems.

But phone hardware and mobile operating systems aren't all that have changed in the last decade. It's now an online world where users are in nearly constant contact with friends, co-workers, family, that high school buddy they haven't seen in 20 years, and random followers they've never met. Applications that once worked only with local copies of documents and data are now interacting with services running in the cloud. And with all this online presence and exposure, security is extremely important. It's no longer acceptable to give software full access to hardware, or to data stored in the file system.

Application development platforms and paradigms have changed as well. With the rise of web applications, a whole new style of application development came into power. Rich interactive applications are the norm, complete with animations, dynamic transitions, and cool graphics. User interfaces are no longer built by developers, but are created by designers who use a whole different set of tools.

Microsoft set out to build a new Windows Phone operating system designed to meet the demands of the altered smartphone market. Microsoft realized they would need a new operating system, backed by a reliable hardware platform, to compete with Apple and Android.

1.2 Windows Phone foundations

Every application developer must understand the hardware and software platforms where their code will run. This is true if you're building desktop applications, web services, or mobile applications. When building Windows Phone applications, you should understand the hardware specifications and know how much memory you can expect to be installed, as well as the supported screen resolutions. Windows Phone provides a unique look and feel that developers should respect when designing user interfaces. You should also know how to leverage or extend the features of built-in applications and services. In this section we talk about the Windows Phone hardware specifications, user interface look and feel, native applications, and the platform APIs you will use to build your own applications.

1.2.1 Hardware specs

With the redesign of the operating system, Microsoft has taken the opportunity to define clear hardware specifications for Windows Phone 7 devices. All devices must meet the minimum hardware requirements.

On Windows Phone 7, all devices have the same screen resolution of 800 x 480 pixels. The physical screen dimensions will also be similar across all devices. A common screen size and resolution allows the same user interface to be reused across different Windows Phone devices.

All Windows Phone devices will provide the user a full four-point multi-touch experience. The operating system provides a *software-based input panel (SIP)* to enable text input for devices without a physical keyboard. Of course, phone manufacturers can add additional user input mechanisms, such as a landscape or portrait physical keyboard, but extra hardware won't be allowed to add extra features to the standard typing. The touch screen is capacitive to give the best experience possible on a mobile device.

Windows Phone 7 devices come with an accelerometer, a compass, and an optional gyroscope. Developers access the raw data from each sensor or use the motion sensor APIs, which wrap up all three sensors into a simple-to-use library. The operating system detects when a device has been rotated from portrait to landscape orientation. The sensors can also be used as an input mechanism for controlling an application or game. The sensors are covered in more detail in chapters 8, 13, and 15.

An FM radio is a mandatory requirement for Windows Phone 7 devices. A user can access the radio from the Zune application in the Music + Videos Hub, but developers can also create a customizable FM radio player using the `FMRadio` class in the `Microsoft.Devices.Radio` namespace. Programming the FM radio is demonstrated in chapter 7.

The minimum hardware specifications also require the following:

- An Assisted GPS receiver to enable location-aware applications
- A camera having a minimal resolution of 5 Megapixels
- A GPU supporting DirectX 9 acceleration
- Either an 800 MHz or a 1 GHz ARMv7 CPU
- A minimum of 256 MB of RAM and 8 GB of Flash storage

The Windows Phone hardware specification requires certain hardware buttons to be present. Many of these keys are not exposed to developers, and applications cannot detect when they are pressed. The physical buttons which will be mandatory for all Windows Phone devices are

- Volume Up
- Volume Down
- Back
- Start
- Search
- Camera
- Power On/Off

A minimum hardware specification has simplified the task of developing a Windows Phone application. These common hardware specifications have allowed Microsoft to create an emulator that will cover most of the possible user interactions with the device, so that you can test most experiences in your emulator.

Microsoft defined a clear hardware specification to ensure users and developers have the same experience on every device. Microsoft also designed a new user interface to provide a clean look and feel.

1.2.2 A new user interface

Windows Phone has completely redesigned the user interface moving from an icon-centric style to the new graphical interface previously developed for the Zune HD media player. Microsoft designers spent some time looking for a proper way to present content and realized an intuitive style already existed. Signage and typography in railway or metro stations, shown in figure 1.1, are concise ways to present information to people coming from different cultures. Why not port this concept to Windows Phone?

The second pillar of the user interface is full-touch support. The success of devices implementing a full-touch user interface is due to the immediacy provided by a natural way of interacting with applications. Concise indications and full-touch come to play an important role in developing applications as you must align to these concepts when you design your user interface.

One well-known defect of the applications written for Windows Mobile was the lack of a common user experience. We've seen applications very aligned to the template generated by Visual Studio but implemented with a user interface that was built to match the iPhone user experience. This is confusing to the user, and you should

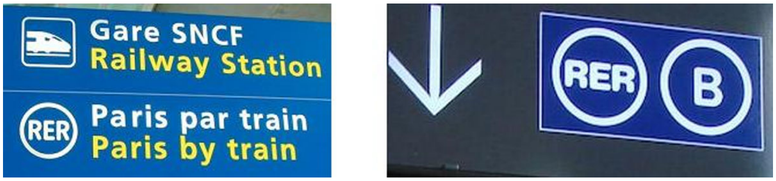


Figure 1.1 Common signs in railways and airports. On the left are icons integrated with text, while on the right only icons are used.

make every effort to match your creations to the Metro design language adopted by the native Windows Phone applications.

Last but not least, when developing your application, you want to target as many users or customers as possible. Globalizing an application doesn't mean just making it right in terms of functionality, but also in terms of contents. We strongly recommend avoiding expressions or icons that don't have a global meaning. Also remember that your application will be inspected by Microsoft prior to publishing it to the Marketplace. There are Marketplace guidelines about what content can and cannot be presented through a Windows Phone application.

1.2.3 User experience

Understanding the user experience of the Windows Phone is important to building an application that feels like it belongs on the phone. The built-in applications, called *hubs*, establish the look and feel of the device and provide integration and extensibility points for third-party applications.

NOTE Only the start experience and the application list are accessible on the emulator.

The hubs are built with two new UI controls named `Panorama` and `Pivot`. You can read more about using the Silverlight versions of `Panorama` and `Pivot` in chapter 10.

START EXPERIENCE

The *Start Experience* is the home screen for Windows Phone. It's the screen displayed when the phone is started. When the user presses the Windows button, they are brought back to the start screen. A user can pin their favorite applications, games, and contacts to the start screen so that they can launch them quickly.

The images displayed on the start screen are named *tiles*. Tiles can be dynamic, displaying information relevant to an application. The tile for the Weather Channel application updates with the latest weather conditions. Other tiles are badged when notifications are ready to be viewed. The tiles for email display a count of new mail messages. The image and title that appear in the start screen are provided by the developer.

Applications can pin multiple tiles to the start screen, each launching to a different spot within the application. Tiles can be updated from code running on the phone, or remotely using the Microsoft Push Notification Service. See chapters 2 and 9 for more details on tiles.

APPLICATION LIST

The *Application List* is where all native and third-party applications appear. It doesn't matter whether the application is built using Silverlight or XNA, or is a native application built by Microsoft, the device vendor, or the mobile carrier. The developer determines the application title and icon that are shown the application list. Games aren't listed in the application list.

GAMES HUB

If your project is declared to be a game, it'll be listed in the *Games Hub* instead of the Application List. The Games Hub is divided into several areas:

- The Collection view lists the games installed on the device.
- The Spotlight view displays news from Xbox Live.
- The Xbox Live view provides access to the user's Xbox Live gamer profile.
- The Requests view lists Xbox Live invitations, messages and notifications.

The game title and icons displayed in the collection are declared by the game developer.

MUSIC + VIDEO HUB

The Music + Video Hub is the central place where you can find all music, video, and podcast activity on the device. The Music + Videos hub is divided into four areas:

- Zune is the central point for playing music, videos, podcasts, and radio, as well as the Zune Marketplace.
- History contains the list of music, videos, playlists, artists, podcasts, and FM radio stations that you recently played. This includes media played by third party applications that integrate with the hub.
- New contains the list of new music, videos, or podcasts that you synced to the phone or downloaded from Zune Marketplace. Third-party applications can add items to the New view.
- Apps contains the list of Music + Videos hub applications that are installed on the device. Third-party media applications are listed here.

The Music + Video Hub provides a few integration points to third-party applications. You can read more about the Music + Video Hub in chapter seven.

PICTURES HUB

The Pictures Hub is the place where you can see all of your photos from different sources. All photos that you took with your mobile phone, synced from the computer, downloaded from the internet, or opened in email will be included in the Pictures Hub. The Pictures Hub is integrated with Windows Live and Facebook, and all photos that you uploaded to those websites will be displayed in the Pictures Hub as well. It also shows the latest photos of your friends in Facebook.

The Pictures Hub can be extended by third-party applications that implement phone editing or sharing features. Extending the Pictures Hub is described in chapter 7.

PEOPLE HUB

The People Hub is the contacts application for Windows Phone. Here's where you find the list of contacts, along with their phone numbers and addresses. The People Hub also displays the latest status and activity obtained from Windows Live and Facebook. Third-party applications can read data directly from the contacts database, and can read and write contacts data with *launchers* and *choosers*, which are introduced in the next section.

Unlike the other hubs, the People Hub is not extensible by Windows Phone applications. The People Hub can be extended by registering new activity streams with the user's Windows Live account. *Activity streams*, a format for syndicating data from social networking applications, are beyond the scope of this book. You can read more about activity streams by visiting http://activitystrea.ms.

Understanding Windows Phones hubs and how they can be extended is key for building applications that enhance user productivity and are integrated with the operating system. Third-party integrated applications and extensions build on top of the features exposed in the platform APIs and frameworks.

1.2.4 Platform APIs and frameworks

Applications run in a sandbox and can't use native APIs, communicate with other processes, or read from the file system. These security measures limit the ability to integrate with native applications and databases. To ease these limitations, native applications also expose various integration points. These integration points come in the form of *launchers*, *choosers*, and *extensions*. The platform also provides access to network APIs so that applications can use web services external to the device. Finally, facilities such as location and notification services are available to third-party developers.

LAUNCHERS

Launchers allow your code to activate a native or built-in application. Data can be passed to the launched application. When the native application is launched, your application is deactivated. Launchers are provided to activate the phone dialer, media player, web browser, and other native applications. Launchers are the only way to initiate a phone call or send an SMS. Launchers are covered in depth in chapter 4.

CHOOSERS

Choosers return data to an application. Choosers are provided to retrieve email addresses, phone numbers, physical addresses, and photographs. Choosers also launch a native application, resulting in the deactivation and/or termination of your application. Choosers are also covered in chapter 4.

EXTENSIONS

Extensions allow an application to integrate their features seamlessly into a native application. For example, the Pictures Hub allows photo editing applications to be launched from its Apps list and from the share and apps menus. The Music + Video Hub allows applications to appear in its Apps list.

NETWORKING

Windows Phone provides HTTP and sockets network communication. HTTP communication is implemented in the `WebClient`, `HttpWebRequest`, and `HttpWebResponse` classes found in the `System.Net` namespace. TCP and UDP communications are implemented with the `Socket` class in the `System.Net.Sockets` namespace. Networking is covered in depth in chapter 9.

NOTIFICATIONS

The Microsoft Push Notification Service provides an API where a phone user can subscribe to a set of custom events. The notification events are defined by third-party applications and must be sent from a dedicated web service implemented by the application developer. Notifications are displayed to the phone user either on the application's tile in the start experience, at the top of the screen as a toast notification, or within the running application. We show you how to build a notification application in chapter 9.

LOCATION

The Location service uses data from the wireless and cellular networks and GPS to allow you to create location-aware applications. Calls to the location cloud service are abstracted behind the `GeoCoordinateWatcher` class in the `System.Device.Location` namespace. In chapter 13 we show you how to use `GeoCoordinateWatcher`.

CUSTOM WEB SERVICES

Beyond providing access to business application data or social networks, custom web services can be used to overcome some of the limitations of phones. If you have a suite of applications that share data, you can use a web service to share the data between them.

1.2.5 AppHub and the Windows Phone Marketplace

AppHub is the portal where Windows Phone and Xbox Live Indie Game developers can find the tools and resources for building and selling applications and games. The AppHub is where you can download the developer tools. You can also find sample code, tutorials, and documentation. If you need advice on a tricky problem, you can submit a question to the developer forums on the AppHub. The AppHub is located at http://create.msdn.com.

Before you can deploy and debug your application on a real phone, or publish your application to the Windows Phone Marketplace, you must purchase a yearly subscription to the AppHub. Depending on what you're building, you might consider waiting to purchase an AppHub subscription until your application is nearly complete, using the emulator to build and test your application.

TIP College students receive free AppHub subscriptions through the *DreamSpark* program. DreamSpark is a Microsoft program providing students with free copies of retail development tools and servers. You can learn more about DreamSpark at http://dreamspark.com.

Once the application has been developed, it must go through an approval process run by Microsoft before being published to the Windows Phone Marketplace. This will ensure that the application conforms to Microsoft requirements for a Windows Phone 7 application. Microsoft's requirements are detailed in the document *Application Certification Requirements for Windows Phone* available from the AppHub and MSDN. More details about marketplace registration are provided in the appendix.

1.3 Comparing Windows Phone to other mobile platforms

This book is written primarily for developers who have some experience working with C# and Silverlight. We focus on the features and APIs that have been introduced specifically for the phone, or have been modified to fit the phone's unique characteristics.

If you already use Silverlight to develop applications, you know it has matured rapidly over the last few years. Silverlight's success as a lightweight application framework makes it ideal to use as the application framework on the mobile device. The Silverlight Framework is rich in features and has been proven with browser and desktop applications. You'll find many of the familiar features and tools. The Windows Phone version of Silverlight is version 4.

NOTE The initial version of Windows Phone 7 used Silverlight 3. Silverlight 4 shipped with Windows Phone 7.5.

If you've used XNA Game Studio, than you know that XNA is built to run on the Xbox, Windows, and the Zune—Windows Phone is just one more platform. Existing developers can easily build and port games for the new devices. Windows Phone introduces a new game development model by integrating Silverlight with XNA, which we introduce in the final section of the book.

If you're not already a Silverlight developer, don't despair. The appendix includes a quick primer for Silverlight and Manning has published several books on C# and Silverlight, which you can find at http://mng.bz/44nv.

But what if you're coming to Windows Phone from some other background? How does the Windows Phone differ from Windows Forms on Windows Mobile? Where do you begin when porting your iOS or Android application? In this section we get you started with Windows Phone development by identifying the similarities and differences with other application platforms.

1.3.1 Windows Mobile

If you're a third-party Windows Mobile developer, then you should know that Windows Phone 7 is not Windows Mobile. You can't use C++ or the Win32 API. If you were hoping that Windows Phone 7 would be backward-compatible with Windows Mobile, then you're out of luck. You may have heard that there is a native SDK, but for now, only device manufacturers, mobile operators, and other special partners get to use it.

Windows Mobile has been a popular operating system because of its extreme customization. Windows Phone is a new operating system and not an upgrade, and applications written for Windows Mobile and Windows Phone 6.5 aren't compatible with Windows Phone 7. Windows Mobile development environments and tools are also incompatible. In this section we illustrate the major changes which will impact every developer with previous experience in Windows Mobile development, starting with the user interface.

BUILDING YOUR INTERFACE

Windows Mobile applications are built with C/C++ and low-level API calls. Neither of these options is available to the Windows Phone developer, who must now use Silverlight and *Extensible Application Markup Language (XAML)*. XAML is a user interface design language first introduced with the Windows Presentation Foundation (WPF) and is a core component of Silverlight. XAML enables separation between the user interface and the code that implements application logic.

DRAWING ON THE SCREEN

Windows Mobile provided two native APIs for drawing text and graphics to the screen:

- Graphics Device Interface (GDI)
- DirectX

Both the APIs are low-level and have a steep learning curve for the standard developer. Being native libraries, neither GDI nor DirectX can be called from managed code running on Windows Phone. The XNA Framework is the managed alternative to DirectX, implementing many of the features available in the DirectX libraries. Silverlight makes use of DirectX and your application will be hardware-accelerated behind the scenes.

CHANGES IN THE USER EXPERIENCE

The Today Screen has been the traditional Windows Mobile shell or system UI. Windows Mobile allows the system shell to be replaced by custom user interfaces built by device manufacturers and third-party developers. Windows Phone provides a new simplified user interface that can't be replaced or modified. The simplified user interface has removed some traditional controls, while introducing new ones designed for touch interaction and to simplify creating user interfaces.

SOFT KEYS SUPPORT

One change you need to keep in mind if you're porting a Windows Mobile application to Windows Phone is the full lack of soft keys, including the hardware buttons associated with them. Another change in the user interface is the menus: they're now basic and most of them are no more than a list.

CHANGES IN THE API

The biggest strength of Windows Mobile was probably its broad compatibility in terms of the programming paradigm and APIs with Windows desktop. This meant that every Windows desktop developer was a potential Windows Mobile developer. On the other hand, Windows Mobile compatibility with the Win32 API brought an additional complexity to the application.

MEMORY MANAGEMENT

A major problem with Windows Mobile applications was the possibility of memory leaks. Because C/C++ requires code to manage its own memory, if the developer allocates memory but forgets to release it during the execution, memory is lost until the process is terminated. Managed applications written in C# or Visual Basic use the .NET

Compact Framework's garbage collector, which is an invisible helper taking care of memory management.

ACCESS TO THE FILE SYSTEM

Windows Mobile applications have almost full access to all the files available on the file system. This capability is useful when developing document centric applications such as a text editor, so that the user will be able to open a file on the file system regardless of its location. On the other hand, a malicious application could corrupt the file system and prevent other applications from being executed, or sniff out sensitive data.

For this reason, each Windows Phone application is locked into a sandbox and can only access files in a reserved portion of persistent memory named isolated storage. There's no way for a Windows Phone application to access data contained in isolated storage belonging to a different application. Isolated storage is covered in chapter 5. Applications requiring access to the whole file system cannot be developed under Windows Phone 7.

MULTITASKING

The Inter-Process Communication (IPC) API of Windows Mobile allows different processes to synchronize with each other using the operating system primitives. Sometimes this was useful as Windows Mobile is a multitasking operating system.

Windows Phone doesn't support true multitasking, at least for applications developed in XNA or Silverlight. Fast application switching allows multiple applications to be resident in memory, but only the foreground application is running, with the background applications remaining in a dormant state. Applications can use background agents to perform limited types of work when an application isn't in the foreground. Fast application switching and background agents are described in chapter 3.

One new possibility for mobile developers, previously available only to desktop developers, is the thread pool. As the creation of a thread is an expensive process and most of the threads are usually blocked on some event, a set of threads is provided by the operating system which will be automatically re-used during the execution. All this is provided for free by the system; in addition to being easy to use, it's a good practice when designing for new systems that could embed multi-core processors. A thread pool automatically scales to multi-core processors without need of code rework.

As you can see, Windows Phone 7 is a completely different platform from Windows Mobile 6. The work required to port existing Windows Mobile applications is no different from that required to port iOS or Android applications.

1.3.2 Apple iOS

At first glance, you might think there's little in common between developing applications for an iOS device and the Windows Phone. On one platform you use Objective-C to write native applications; on the other you use C# to write managed applications. It's our opinion that programming languages and frameworks are just tools in a developer's tool belt, and good developers make use of several languages and frameworks.

If you look beyond the languages and development environments, many of the fundamental concepts exist on both platforms.

Apple and Microsoft both provide free development tools complete with device simulators. Each platform has a set of style guides that applications should adhere to, and also requires a fee-based subscription in order to deploy an application to an actual device. Each platform has a certification process and application store.

BUILDING YOUR INTERFACE

One thing to keep in mind when porting an iOS application is the differences in the user interface guidelines. You shouldn't build an application with an iOS look and feel for the Windows Phone. An iOS application ported to Windows Phone will have a different look and feel, user interaction model, and workflow. Don't use chrome and icons from iOS.

Is your application built with controls from UIKit or does it use OpenGL ES? The Silverlight Framework offers many of the controls and widgets provided by UIKit. On the other hand, OpenGL developers will use the XNA Framework to build applications. You can also mix application style widgets from Silverlight with XNA type graphics.

You'll build your Silverlight applications using Visual Studio and Expression Blend. Your views will be built using XAML, an XML-based markup language. XAML can be coded by hand in Visual Studio's text editor, or with the visual editors in Visual Studio and Expression Blend. The core Silverlight Framework, along with the Silverlight Toolkit, provides most of the controls you'll need when building an application.

If your iOS application uses Core Animation, you'll use the animation and storyboard classes from the `System.Windows.Media.Animation` namespace. Learn to use Expression Blend's storyboard editor if you're doing anything beyond very simple animations.

Silverlight applications are navigation-style applications, driven by the `NavigationService`. The `NavigationService` is similar to the `UINavigationController` provided by the iOS framework, and is used to move between different pages or views. The difference is that all Silverlight applications use the `NavigationService`, even the simplest one-page application.

INTERACTING WITH THE NATIVE APPLICATIONS

Like the iOS SDK, Windows Phone provides limited access to the phone dialer, SMS text application, and email. On iOS, the phone dialer is accessed via the `tel` URL; on Windows Phone you use the `PhoneCallTask`. `MFMessageComposeViewController` and `MFMailComposeViewController` are replaced by `SmsComposeTask` and `EmailComposeTask`.

The iOS SDK provides access to the address book with several classes in the Address Book and Address Book UI frameworks. On Windows Phone, read-only access to the address book is exposed via classes in the `Microsoft.Phone.UserData` namespace. Developers can also interact with the contacts database via a few launchers and choosers. You can prompt the user to choose a phone number, email address, or physical address with `PhoneNumberChooserTask`, `EmailAddressChooserTask`, and `AddressChooserTask`. You can prompt the user to save a phone number or email address with

`SavePhoneNumberTask` and `SaveEmailAddressTask`. You can read more about launchers and choosers in chapter 4.

USING THE SENSORS

Like the iPhone, the Windows Phone has an accelerometer, a compass, and a camera. Some Windows Phones will also have a gyroscope. The initial release of Windows Phone didn't provide an API to access the compass, and access to the camera was limited. The Windows Phone SDK 7.1 introduced new APIs providing access to the compass, gyroscope, and the camera. Using the `CameraCaptureTask`, you can launch the camera UI and manipulate a photo taken by the user. You can take direct control of the camera by using either the `PhotoCamera` or the `WebCamera` APIs. Working with the camera is covered in chapter 6.

The Windows Phone complement to `UIAccelerometer` is the `Microsoft.Devices.Accelerometer` class. The `Compass` class is the Windows Phone equivalent to `CLHeading`. Motion detection features available by the Core Motion framework are provided by the `Gyroscope` and `Motion` classes. We show you how to use the accelerometer, compass, and gyroscope in chapter 8.

STORING DATA

An iOS application can store its data in user defaults, on the file system, or in a database. The iOS SDK makes use of SQLite for local database management.

Windows Phone does provide limited access to the file system. An application can only write files to isolated storage, and it doesn't have access to any other part of the file system. Isolated storage is similar to an iOS application's Documents folder.

Another way to store data is with the `IsolatedStorageSettings` class. This class is similar to the `NSUserDefaults` class in the iOS framework. It's intended to be used to store lightweight data objects and is ideal for storing user preferences. One difference between `NSUserDefaults` and `IsolatedStorageSettings` is that `IsolatedStorageSettings` isn't global, and settings can't be shared between different applications.

Applications can store data in a Microsoft SQL Server Compact (SQL CE) database using the LINQ to SQL framework. SQL CE is a lightweight database engine designed to run on mobile devices. The database files are written to a special folder in isolated storage, and can't be shared with other applications. Chapter 5 demonstrates how to use each of the data storage options in your applications.

MEDIA

The iPhone uses the iPod software to play audio and video files. The iOS SDK's Media Player framework allows developers to access the library of music and videos, and to play them inside their applications. The Windows Phone uses Zune for its media library, shown to users in the Music + Videos Hub. Applications can play audio and video files with the `MediaPlayerLauncher` class. Developers can also access the Zune library using the classes in the `Microsoft.Xna.Framework.Media` namespace. The `MediaPlayer` class can be used to play songs, whereas the videos are played with the `VideoPlayer` class.

Silverlight applications can use the XNA Media framework, but Silverlight also has its own media controls in the `System.Windows.Media` namespace. The `MediaElement` control supports audio and video playback. The `MediaStreamSource` class can be used to manipulate audio and video playback or implement custom media containers.

The Windows Phone equivalent to the iOS's `AVAudioRecorder` class is the `Microsoft.Xna.Framework.Audio.Microphone` class.

Your application can integrate into the Music + Video Hub on the phone. Your application can be listed in the hub's Apps list, and media played by your application can be shown in the Hub's History page.

You can read about working with media, the microphone, and the Music + Videos Hub in chapters 7 and 12.

NETWORKING

The iOS SDK offers several classes to enable network programming. A developer can choose to program using raw sockets, or higher-level protocols such as HTTP and FTP. Windows Phone offers sockets and HTTP support. You perform HTTP communication using the `HttpWebRequest`, `HttpWebResponse`, and `WebClient` classes in the `System.Net` namespace. Sockets programming is performed using classes in the `System.Net.Sockets` namespace.

Microsoft has also built a notification service to allow web services to push notifications to a phone. Developers host their own web service or other application. The application service sends notifications to Microsoft's Push Notification web service, which forwards notification to a user's phone. Interaction with the notification service is covered in chapter 9.

As you can see, there are many differences between the iOS and the Windows Phone. There are also a number of similarities and developers should be able to port most applications to the Windows Phone.

1.3.3 Android

Android is another new mobile operating system that's capturing the hearts and minds of consumers and developers. Like the iPhone, there are many differences and many similarities between Android and Windows Phone. Like Windows Phone, Android runs on a number of different devices, from a number of different manufacturers. Unlike Microsoft, Google hasn't dictated the hardware specifications to the manufacturers and developers must design and test on several hardware configurations.

Android and Microsoft both provide free development tools complete with device emulators. But Microsoft requires a fee-based subscription in order to deploy an application to an actual device and certifies each application before making the application available in the application store.

RUNTIME ENVIRONMENT

Windows Phone applications run in the .NET Compact Framework Common Language Runtime (CLR). The CLR is a virtual machine much like the Dalvik virtual machine that runs on Android. Applications are packaged in .xap files, which is a zip archive of the assemblies and resources in the application bundle.

Windows Phone places restrictions on the types of applications that can run on the phone. Android allows for background services and UI-less broadcast receivers to run on the phone. Though Windows Phone offers limited support for background operations with background agents, there's no counterpart to broadcast receivers. Windows Phone doesn't have system alarms or triggers that can directly start an idle application. Windows Phone applications can be started when the user responds to alarms, reminders, or notifications.

The Android runtime does limit access to certain features with manifest permissions. Windows Phone uses a similar security model by requiring capabilities to be declared in the application manifest.

BUILDING YOUR INTERFACE

Android activities are loosely related to pages in a Silverlight application. Each page of an application has a unique address, and the operating system will use a page's URL to navigate to the page when restarting an application. Developers can use a page's URL when creating tiles. Android programmers declare user interfaces with layout XML files. Silverlight user interfaces are declared using XAML, which are also XML files. If your application makes use of the Android `MapView`, you'll want to read about using the Bing Maps control in chapter 13.

INTERACTIONS WITH OTHER APPLICATIONS

Android applications interact with built-in and third-party applications by dispatching *intents*. Windows Phone applications interact with native applications via launchers and choosers. Windows Phone doesn't allow third-party applications to interact with other third party applications, and developers can't create new launchers or choosers.

Android applications can replace, enhance, or just eavesdrop on another application by handling the same Intents. Windows Phone doesn't allow third-party applications to replace any launchers or choosers. You can enhance the Pictures Hub and the Music + Videos Hub by implementing the required extensibility points.

Android applications share data by exposing and using content providers. On Windows Phone, there's no way to expose your data to other applications, and other applications can't use your data.

You can read about the available launchers and choosers in chapter 4.

STORING DATA

An Android application can store its data in shared preferences, on the file system, or in a database. Android uses SQLite for local database management.

Windows Phone does provide limited access to the file system. An application can only write files to isolated storage, and doesn't have access to any other part of the file system. You can't read another application's files, and other applications can't read your application's files.

Another way to store data is with the `IsolatedStorageSettings` class. This class is similar to `SharedPreferences` in the Android framework. It's intended to be used to store lightweight data objects and is ideal for storing user preferences. One difference between `SharedPreferences` and `IsolatedStorageSettings` is that `IsolatedStorageSettings` is not global, and settings can't be shared between different applications.

Window Phone applications can store data in a Microsoft SQL Server Compact (SQL CE) database using the LINQ to SQL framework. SQL CE is a lightweight database engine designed to run on mobile devices. The database files are written to a special folder in isolated storage, and can't be shared with other applications. Chapter 5 demonstrates how to use each of the data storage options in your applications.

MEDIA

Android uses the OpenCORE library to play and record audio files and to play video files. OpenCORE's `MediaPlayer` class is used to play audio, whereas the `VideoView` widget is used to play video. Windows Phone applications use the `MediaPlayerLauncher` class to play audio and video files. Developers can also access the Zune library using the classes in the `Microsoft.Xna.Framework.Media` namespace. The `MediaPlayer` class can be used to play songs, whereas the videos are played with the `VideoPlayer` class.

Silverlight applications can use the XNA Media framework, but Silverlight also has its own media controls in the `System.Windows.Media` namespace. The `MediaElement` control supports audio and video playback. The `MediaStreamSource` class can be used to manipulate audio and video playback or implement custom media containers.

The Windows Phone equivalent to the Android's `MediaRecorder` class is the `Microsoft.Xna.Framework.Audio.Microphone` class. You can read about working with media, the microphone, and the Music + Videos Hub in chapters 7 and 12.

NETWORKING

Android provides a variety of networking options starting with raw sockets and extending through HTTP. Windows Phone offers sockets and HTTP support. You perform HTTP communication using the `HttpWebRequest`, `HttpWebResponse`, and `WebClient` classes in the `System.Net` namespace. Sockets programming is performed using classes in the `System.Net.Sockets` namespace.

Android networking applications can use the `ConnectivityManager` class to determine the status of the device's network connection. To check the network status of a Windows Phone, you use the `NetworkInterface` class in the `Microsoft.Net.NetworkInformation` namespace.

In many ways, the Android platform is more like the Windows Mobile platform. Applications have fewer restrictions and can replace core features of the operating system. Manufacturers can change the look and feel of the operating system. Developers must build for a wider range of hardware configurations. There are going to be a certain set of applications that can't be ported to Windows Phone because of the limitations enforced by the operating system.

1.4 The Windows Phone Developer Tools

In order to build great applications, you need great development tools. Microsoft's Visual Studio and Expression Blend fit the description. Visual Studio 2010 Express for Windows Phone joins the list of no-cost express developer tools provided by Microsoft. XNA Game Studio has been updated to build Windows Phone games. And a no-cost version of Expression Blend 4 has been made available. All of these tools have been packaged together and are distributed as the *Windows Phone Developer Tools* which can be freely downloaded from the AppHub at http://create.msdn.com.

1.4.1 Visual Studio for Windows Phone

The Windows Phone Developer Tools installs an express edition of Visual Studio 2010 configured with the phone development tools. If you already have a retail edition of Visual Studio 2010 installed on your computer, the phone development tools will be installed as a plug-in to the IDE. Windows Phone projects can be written in both C# and Visual Basic.

We use the Express edition throughout the book for the screen shots and sample code. Code and user interface design features will work the same in the retail editions of Visual Studio 2010.

You can launch the IDE by opening the Start Menu and clicking on Microsoft Visual Studio 2010 Express for Windows Phone in the Microsoft Visual Studio Express folder. Figure 1.2 shows the Visual Studio IDE.

Visual Studio 2010 Express for Windows Phone, from here on referred to as Visual Studio, can be used in two different modes—Basic and Expert. We suggest you enable Expert mode so that you can use all the available features. To enable the Expert mode, you have to select the Tools->Settings->Expert Settings menu option. Expert mode unlocks some toolbars and several menu items.

1.4.2 Expression Blend for Windows Phone

Visual Studio has cool features but it's not so friendly for the user interface designers on your team. Microsoft has created a tool for designers named *Expression Blend*. Originally part of the Expression Studio suite, a no-cost edition of Expression Blend has been provided for creating Windows Phone applications. Expression Blend allows the designer to create user interfaces without writing a single line of code.

Expression Blend can create the same Silverlight projects as Visual Studio. A designer can edit the same solution, project, and code files that a developer edits in Visual Studio. Though we may occasionally cover Expression Blend features in the book, our focus will remain on using Visual Studio. A primer on Expression Blend is available in the appendix.

1.4.3 XNA Game Studio

XNA Game Studio, another Visual Studio add-on, provides a set of tools and libraries that can be used to build games for Windows, Xbox, Zune and now the Window

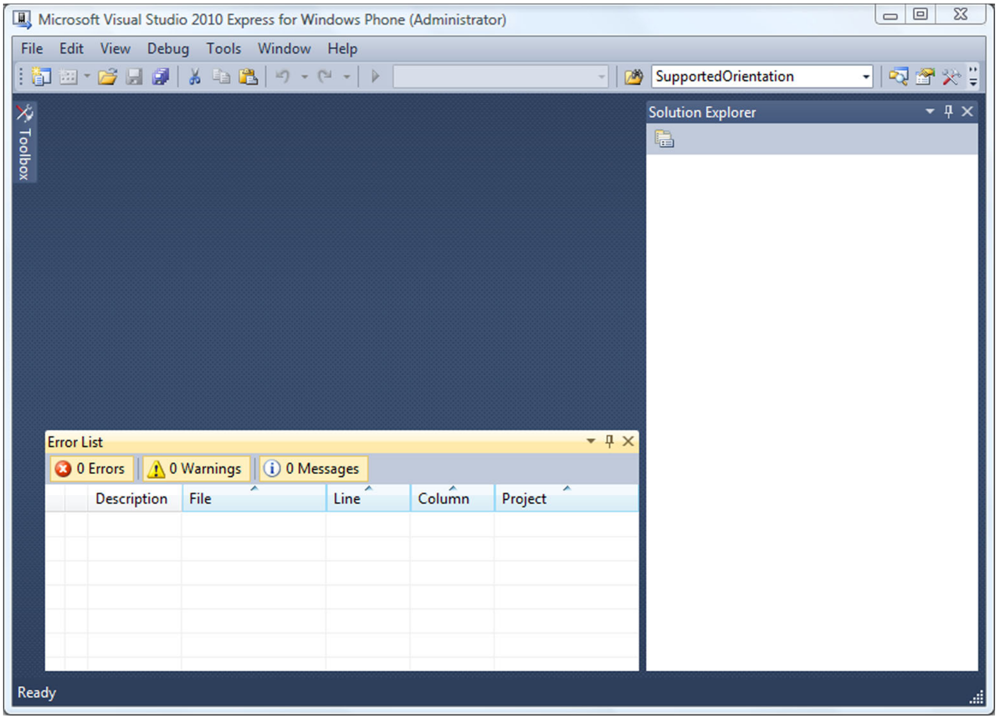

Microsoft Visual Studio 2010 Express for Windows Phone (Administrator)
File Edit View Debug Tools Window Help
SupportedOrientation
Toolbox
Solution Explorer
Error List
0 Errors
0 Warnings
0 Messages
Description
File
Line
Column
Project
Ready

Figure 1.2 **Visual Studio 2010 Express for Windows Phone**

Phone. XNA Game Studio 4 provides the tools necessary for creating Windows and Xbox games, which are beyond the scope of this book.

XNA applications usually import content or assets created by artists. This could be graphics, 3D models, music, or videos. In the last section of this book, we show you how to build rich graphics applications by integrating XNA with Silverlight.

1.4.4 Windows Phone Emulator

The Windows Phone 7 Emulator should work on most computers running the Windows 7 or Windows Vista operating system. System requirements are listed in the MSDN documentation at http://mng.bz/PYCd. The emulator performs better if your computer has a CPU with virtualization extensions like most of the recent AMD and Intel CPUs. The emulator works best if a DirectX 10 or later graphics card with a WDDM 1.1 driver is present.

NOTE You can determine whether your computer has a supported GPU and driver with the DirectX Diagnostics Tool that is part of the DirectX SDK. You can download the DirectX SDK from the DirectX Developer Center at http://mng.bz/wczO.

The DirectX 10 GPU and WDDM driver are mandatory for XNA games, but aren't necessarily required by most Silverlight applications.

The emulator doesn't require special binaries to execute XNA games or Silverlight applications for Windows Phone. The emulator can be used to verify orientation changes in your application by using the rotation buttons on the emulator's command toolbar.

Keep in mind that on the emulator you're sharing the network connection of your PC, so the bandwidth available is greater than what would be available to a real phone. The emulator doesn't allow you to simulate out-of-coverage scenarios or bandwidth changes (such as 2.5G to 3G), and when checking network information the `NetworkInterface` class always returns `WiFi`. In order to verify network connectivity, you can use the full working versions of Internet Explorer available in the emulator.

The Settings application found in the application list can be used to change the emulator's default configuration. But the settings revert to their defaults when the emulator is stopped and restarted. You'll need to change the settings to verify your application behaves appropriately under different configurations and locales. The emulator settings are

- Theme and accent color
- Date and Time
- Region and Language

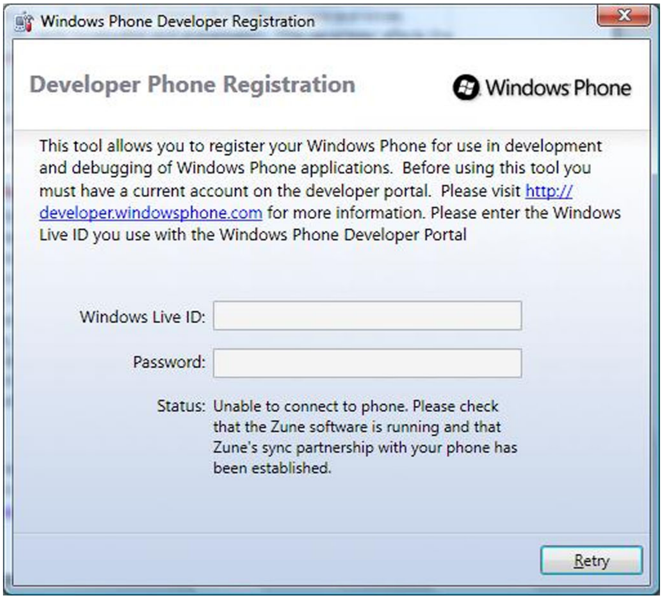
If your computer is running Windows 7 and uses a true multi-touch monitor, the emulator will register touches to the computer monitor. Otherwise, the emulator simulates touches with the mouse. The emulator can also switch between using the SIP and treating your computer's keyboard as a hardware keyboard.

1.4.5 Windows Phone Developer Registration tool

Applications can only be installed onto a phone by the Windows Phone Marketplace. Limited exceptions are made to phones registered to developers who have accounts with the AppHub. AppHub accounts aren't free, and can be purchased from the developer portal at http://create.msdn.com. The Windows Phone Marketplace procedures are covered in more depth in the appendix. Windows Phone applications can't be distributed as standalone packages. In order to develop your own application, you need to enable your device to allow the deployment of XAP files.

Once your account has been verified by the AppHub, you can launch the Windows Phone Developer Registration tool from the Windows Phone Developer Tools folder in your Start Menu. This tool, shown in figure 1.3, will prompt you to enter your AppHub credentials and select a connected phone. You need to plug the device into your PC, pair it to the Zune client, and have your PC connected to the internet in order to connect to Microsoft's registration servers.

There's a limit of three phones that can be registered to a single account. There's also a limit of ten developer applications that can be installed on a phone at the same time. If you reach the installed application limit, you must uninstall one or more



Developer Phone Registration

This tool allows you to register your Windows Phone for use in development and debugging of Windows Phone applications. Before using this tool you must have a current account on the developer portal. Please visit http://developer.windowsphone.com for more information. Please enter the Windows Live ID you use with the Windows Phone Developer Portal

Windows Live ID:

Password:

Status: Unable to connect to phone. Please check that the Zune software is running and that Zune's sync partnership with your phone has been established.

Figure 1.3 Windows Phone Developer Registration tool

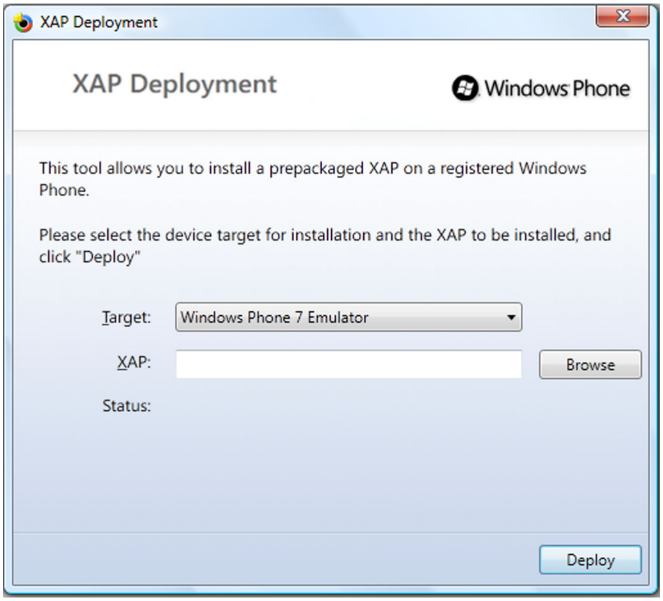
developer applications before you'll be able to deploy a new application from Visual Studio. Occasionally your phone registration will expire and you'll receive an error when attempting to deploy an application to a device. You will then need to reregister the phone with the registration tool.

The Windows Phone Developer Registration tool can also be used to unregister a phone. If you need to unregister a phone, but don't have it available because it has been lost or broken, you can unregister the device from your AppHub account's profile page, accessed through your browser at https://users.create.msdn.com/Account/Profile.

1.4.6 XAP Deployment tool

To support testing application by non-developer team members, you can deploy just the executable binary of a Windows Phone application (the .XAP file) to the emulator or to a registered phone using the XAP Deployment tool. The XAP Deployment tool, shown in figure 1.4, is launched from the Start Menu -> Windows Phone Developer Tools folder. You only need to select the target device and the XAP file, and then click the Deploy button.

When the deployment is complete, the application can be started from the Application List. An application can be uninstalled from the Application List as well. Tap and hold the application's icon until the context menu appears and select the uninstall option.



This tool allows you to install a prepackaged XAP on a registered Windows Phone.

Please select the device target for installation and the XAP to be installed, and click "Deploy"

Target: Windows Phone 7 Emulator

XAP:

Status:

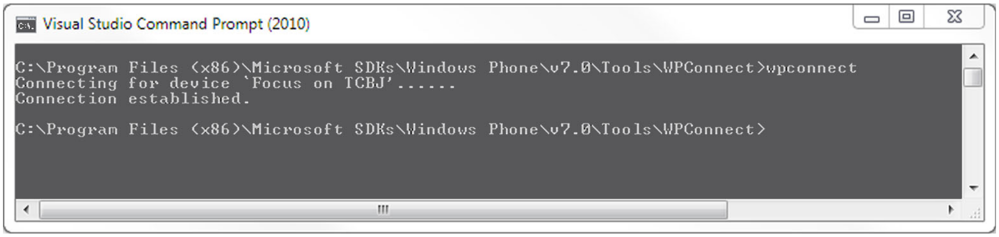
Figure 1.4 Application to deploy a binary (XAP) file to the device

1.4.7 WPConnect

When you use Visual Studio to debug applications running on a real phone, the phone must be connected to your computer. It's not sufficient to have your phone connected via the USB cable; a connection must also be established via software. Usually the Zune software handles this connection.

When a phone is connected to Zune, the phone's pictures and media databases are locked. You'll experience errors if you attempt to debug software that uses these libraries. Microsoft has provided the WPConnect tool to allow Visual Studio to connect a phone without running the Zune software.

The WPConnect tool is installed by the Windows Phone Developer Tools. You can find the tool in %ProgramFiles%\Microsoft SDKs\Windows Phone\v7.1\Tools\WPConnect. Before running WPConnect, you must connect your phone and launch the Zune software. Close the Zune software once you verify that the phone was found. When you run WPConnect you'll see a confirmation message like that shown in figure 1.5.



```
C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.0\Tools\WPConnect>wpconnect
Connecting for device `Focus on TCBJ'......
Connection established.

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.0\Tools\WPConnect>
```

Figure 1.5 The WPConnect confirmation message

Of course, you can use this tool when you are debugging any application, not just applications that use the media library.

1.4.8 Isolated Storage Explorer tool

Most applications require some form of data storage—from user preferences and user-created data to local caches of data stored in a cloud application or web service. Each application is allotted its own storage sandbox on the phone, isolated from all other applications and from the operating system. Isolated storage will be empty when an application is first deployed to the emulator or a device. During execution, many applications will store data and settings in isolated storage.

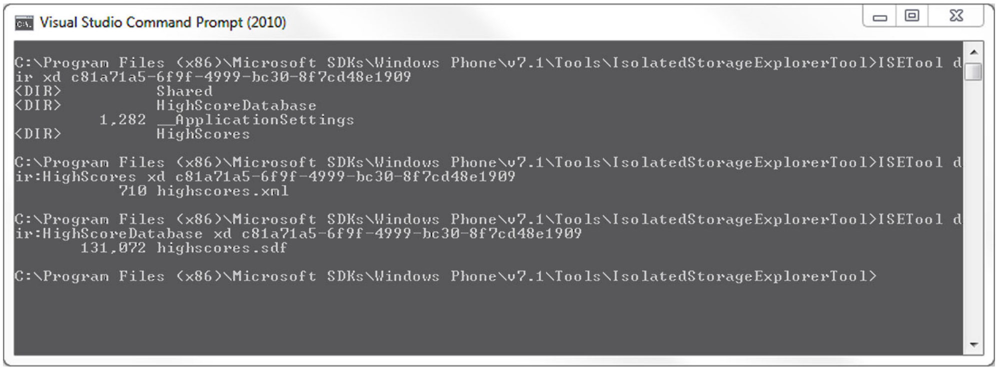
While testing and debugging an application, developers might want to examine the files written to isolated storage or maybe even write data files to isolated storage to facilitate testing. The Isolated Storage Explorer tool (ISETool) is included in the Windows Phone 7.1 SDK to enable these scenarios. The ISETool allows a developer to take a snapshot of an application's isolated storage, copying the files from the phone to a desktop folder. The ISETool can also be used to copy files from the desktop to an application's isolated storage folder. The ISETool will also list the files in an isolated storage folder, as shown in figure 1.6.

The ISETool requires the application's product GUID. The product GUID is generated by the Visual Studio project templates and is declared in a project's application manifest file, which is named WMAppManifest.xml. You'll learn more about the application manifest in the next chapter.

We show you how to use the Isolated Storage Explorer tool to populate a read-only database in chapter 5.

1.4.9 Marketplace Test Kit

Once an application has been developed, it must go through an approval process run by Microsoft before being published to the Windows Phone Marketplace. This will ensure that the application conforms to Microsoft requirements for a Windows Phone



```
C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool>ISETool d
ir xd c81a71a5-6f9f-4999-bc30-8f7cd48e1909
<DIR>          Shared
<DIR>          HighScoreDatabase
         1,282 __ApplicationSettings
<DIR>          HighScores

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool>ISETool d
ir:HighScores xd c81a71a5-6f9f-4999-bc30-8f7cd48e1909
           710 highscores.xml

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool>ISETool d
ir:HighScoreDatabase xd c81a71a5-6f9f-4999-bc30-8f7cd48e1909
       131,072 highscores.sdf

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool>
```

Figure 1.6 Using the Isolated Storage Explorer Tool to list the files in isolated storage in the emulator

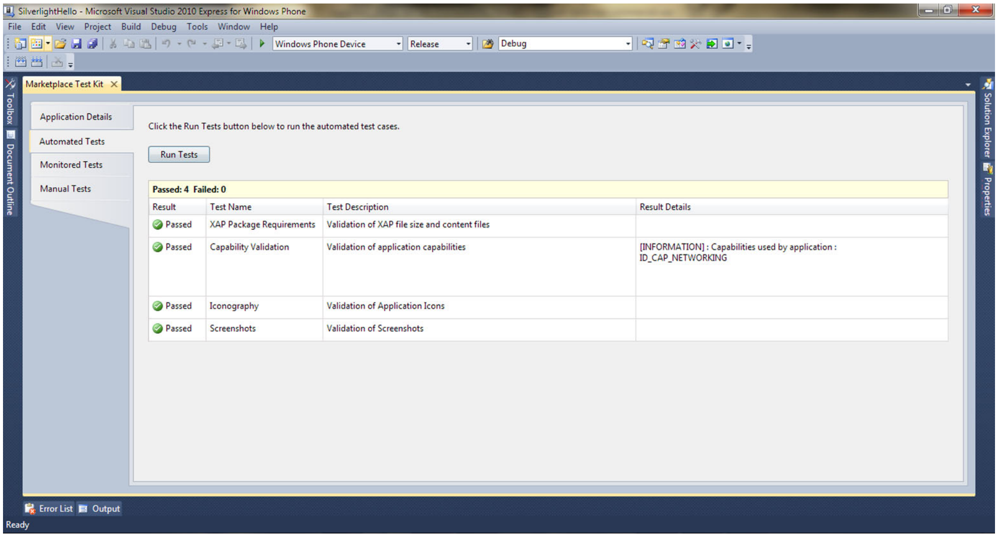
application. Microsoft's requirements are detailed in the *Application Certification Requirements for Windows Phone* available from http://create.msdn.com.

The Marketplace Test Kit includes a series of automated, monitored, and manual tests you can use to ensure your application meets the *Application Certification Requirements for Windows Phone*. The Marketplace Test Kit also helps you assemble the graphics and screen shots that will be submitted along with the application's .xap file. The Marketplace Test Kit is installed when you install the Windows Phone Developer Tools. A screen shot of the Marketplace Test Kit is shown in figure 1.7. The Marketplace Test Kit is an extension to Visual Studio and is accessed from the *Open Marketplace Test Kit* option in the Visual Studios Project menu.

If your application violates any of the tests, it will be rejected during the submission process. Nobody likes to receive rejection letters. Though you should run the ingestion tool before you submit your application, you may also choose to run it periodically during development so that you can detect and fix issues as early as possible.

The Capability Validation test reports the security capabilities required by an application.

Windows Phone is all about security sandboxes and user disclosure. Security capabilities are one face of the operating system's security model and a Windows Phone application must declare which capabilities or features of the operating system it uses. The capabilities used by an application are declared in the application's manifest file. When an application is submitted to the marketplace, the certification process inspects the compiled code and updates the manifest with the discovered capabilities. Table 1.1 details the set of capabilities that can be listed in the manifest.


SilverlightHello - Microsoft Visual Studio 2010 Express for Windows Phone
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Windows Phone Device
Release
Debug
Toolbox
Document Outline
Marketplace Test Kit
Application Details
Automated Tests
Monitored Tests
Manual Tests
Run Tests
Solution Explorer
Properties
Error List
Output
Ready

Click the Run Tests button below to run the automated test cases.

Passed: 4 Failed: 0

Result	Test Name	Test Description	Result Details
✓ Passed	XAP Package Requirements	Validation of XAP file size and content files	
✓ Passed	Capability Validation	Validation of application capabilities	[INFORMATION] : Capabilities used by application : ID_CAP_NETWORKING
✓ Passed	Iconography	Validation of Application Icons	
✓ Passed	Screenshots	Validation of Screenshots	

Figure 1.7 The results of automated tests performed by the Marketplace Test Kit, including a list of security capabilities used by the application

Table 1.1 Security capabilities

Capability ID	Description	Required by
`ID_CAP_APPOINTMENTS`	Access appointment data from the calendar	`Microsoft.Phone.UserData.Appointments`
`ID_CAP_CONTACTS`	Access contact data from the address book	`Microsoft.Phone.UserData.Contacts`
`ID_CAP_GAMERSERVICES`	Use Xbox Live APIs	`Microsoft.Xna.Framework.GamerServices`
`ID_CAP_IDENTITY_DEVICE`	Access to device information	`Microsoft.Phone.Info.DeviceExtendedProperties`
`ID_CAP_IDENTITY_USER`	Access to user information	`Microsoft.Phone.Info.UserExtendedProperties`
`ID_CAP_ISV_CAMERA`	Access the Camera API and the raw image stream	`Microsoft.Devices.PhotoCamera`
`ID_CAP_LOCATION`	Use location services	`System.Device.Location`

`ID_CAP_MEDIALIB`	Access the media library	`Microsoft.Devices.MediaHistory`
		`Microsoft.Devices.Radio.FMRadio`
		`Microsoft.Xna.Framework.GamerServices`
		`Microsoft.Xna.Framework.Media`
		`System.Windows.Media.MediaStreamSource`

`ID_CAP_MICROPHONE`	Record with the microphone	`Microsoft.Xna.Framework.Audio.Microphone`

`ID_CAP_NETWORKING`	Use network services	`Microsoft.Phone.Controls.WebBrowser`
		`Microsoft.Phone.Notification`
		`Microsoft.Xna.Framework.GamerServices`
		`System.Net`

`ID_CAP_PHONEDIALER`	Initiate phone calls	`Microsoft.Phone.Tasks.PhoneCallTask`
`ID_CAP_PUSH_NOTIFICATION`	Receive push notifications	`Microsoft.Phone.Notification`
`ID_CAP_SENSORS`	Use the Accelerometer	`Microsoft.Devices.Sensors`
`ID_CAP_WEBBROWSERCOMPONENT`	Use the web browser control	`Microsoft.Phone.Controls.WebBrowser`
`ID_HW_FRONTCAMERA`	Access the forward facing camera	`Microsoft.Devices.PhotoCamera`

The manifest file created by the Visual Studio project templates automatically declares every capability except `ID_HW_FRONTCAMERA`. The developer can remove any of the capabilities that aren't required for their application. During marketplace certification, the list provided by the developer is deleted and replaced by a list of capabilities detected by the certification tools. The assembly is examined for calls to the secured APIs and when one is found, the matching capability is re-added to the manifest. If required capability isn't listed in the manifest, the secured API will throw a `UnauthorizedAccessException`. When an application is downloaded from the marketplace, the list of capabilities used by an application is displayed to the user, allowing the user to make an informed decision before purchasing the application.

1.5 Summary

This chapter has been an introduction to the Windows Phone platform. Windows Phone 7 is not an upgrade of Windows Mobile, but is an entirely new operating system. Developers moving to Windows Phone from Windows Mobile, or from desktop applications, must learn to work with the Windows Phone Developer Tools. Windows Phone 7 is locked down pretty tight, and many types of applications simply can't be ported to Windows Phone 7.

You'll see in the next chapter how easy it is to create Windows Phone applications. Hopefully the ease of development mitigates the lack of advanced functionality that many developers have come to expect from Windows-based platforms.

Now, install the development tools and move to the next chapter: it's time to code!

2

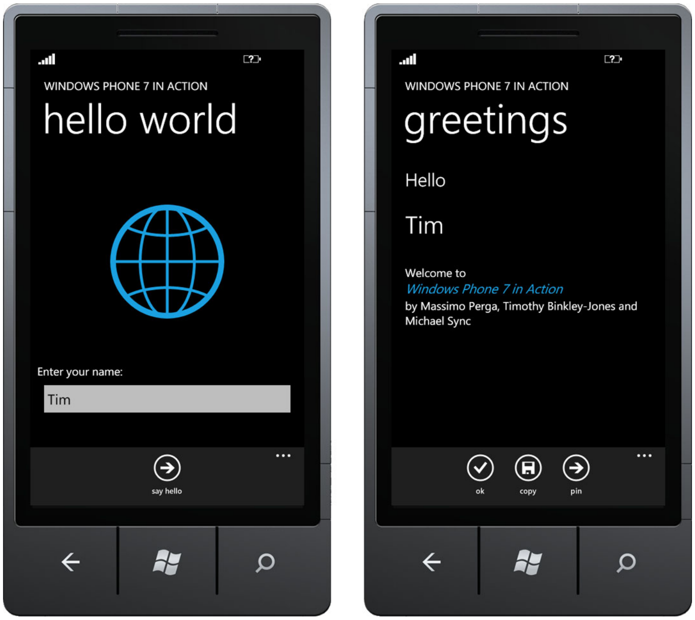
Creating your first Windows Phone application

This chapter covers

- Creating your first Silverlight application
- Handling touch events
- Navigating between pages
- Trial licensing

Now that you have the necessary background on the Windows Phone platform and the Windows Phone Developer Tools, it's time to get down to business and start programming. You'll start by building a Hello World project. For developers experienced with Visual Studio, simple Hello World projects may seem unduly remedial. Windows Phone projects have several unique settings and features that you need to understand to build proper applications and games. The Hello World project in this chapter is designed to highlight these aspects of Windows Phone development.

You'll build a Hello World Silverlight application and explore a few of the phone-specific extensions to Silverlight. Silverlight applications have several project properties unique to Windows Phone. Two of these properties define the icons used in the phone's start screen and Applications List. Other properties determine the titles shown next to the start and application list icons. You'll learn how to use the Visual Studio project templates to generate a new application and how to



Enter your name: Tim

Welcome to *Windows Phone 7 in Action* by Massimo Perga, Timothy Binkley-Jones and Michael Sync

Figure 2.1 The Silverlight Hello World application

use the item templates to generate a new page for your application. You'll also learn how to deploy the application to the emulator or a physical device and use the debugger to step through code.

TIP If you're new to Silverlight development, read the primers for Expression Blend and Silverlight in the appendices.

In most ways, building a Silverlight application for the phone is the same as building one for the browser or the desktop, but there are some minor differences. You'll see some of the differences as you build your application. The Hello World application that you'll create is shown in figure 2.1.

The application displays a title, draws a globe, and prompts the user to enter their name. When the user presses the toolbar button, the application navigates to a greeting page. You'll start building your application by creating a new Silverlight project.

2.1 Generating the project

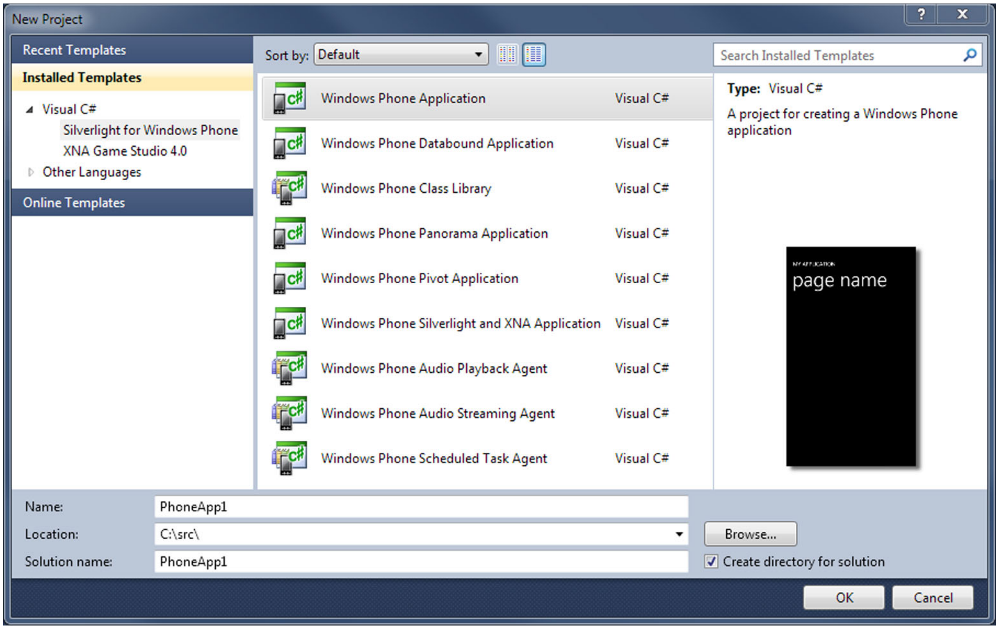
To start the Hello World application, you'll use the Windows Phone Application project template in Visual Studio. The Windows Phone Application project template is just one of the several Silverlight project templates that are installed with Visual Studio. Table 2.1 lists the available project templates.

You'll get started by opening Visual Studio and creating a new project. Figure 2.2 shows the new project dialog for the Hello World Silverlight application. Name the project SilverlightHello.

Table 2.1 Windows Phone project templates

Project template	Description
Windows Phone Application	A basic application skeleton with a single page.
Windows Phone Databound Application	An application demonstrating page navigation, databound list controls, and the MVVM pattern.
Windows Phone Class Library	A simple library for creating reusable components.
Windows Phone Panorama Application	An application demonstrating a databound `Panorama` control and the MVVM pattern. The `Panorama` control is covered in chapter 10.
Windows Phone Pivot Application	An application demonstrating a databound `Pivot` control and the MVVM pattern. The `Pivot` control is covered in chapter 10.
Windows Phone Silverlight and XNA Application	An application that mixes Silverlight and XNA Framework graphics. You'll build an application that uses both Silverlight and XNA in chapter 14.
Windows Phone Audio Playback Agent	A library containing an application's background audio logic. Audio Playback Agents are covered in chapter 7.
Windows Phone Audio Streaming Agent	A library containing an application's background streaming audio logic.
Windows Phone Scheduled Task Agent	A library containing an application's background processing logic. Scheduled Tasks Agents are covered in chapter 3.

Each of the project templates listed here are available for both C# and Visual Basic projects.



Windows Phone Application	Visual C#
Windows Phone Databound Application	Visual C#
Windows Phone Class Library	Visual C#
Windows Phone Panorama Application	Visual C#
Windows Phone Pivot Application	Visual C#
Windows Phone Silverlight and XNA Application	Visual C#
Windows Phone Audio Playback Agent	Visual C#
Windows Phone Audio Streaming Agent	Visual C#
Windows Phone Scheduled Task Agent	Visual C#

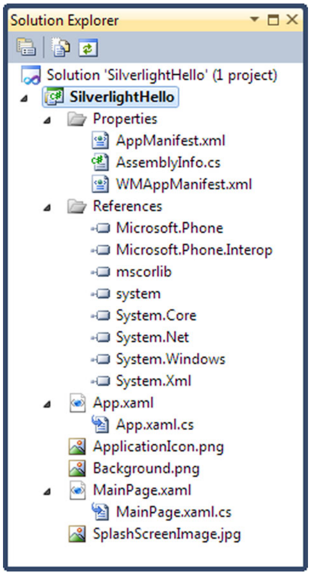
Type: Visual C#

A project for creating a Windows Phone application

Name: PhoneApp1
Location: C:\src\
Solution name: PhoneApp1
☑ Create directory for solution

Figure 2.2 Visual Studio's New Project dialog box

Once you click OK, you'll be prompted with a dialog asking you to pick the target operating system version. This dialog can be confusing because it lists the Windows Phone SDK versions and not the operating system versions. If you're building an application that makes use of the new features in the Windows Phone 7.5 operating system, choose *Windows Phone OS 7.1* from the drop-down. After you click the OK button, a new Visual Studio solution and project are created. The IDE opens MainPage.xaml in the editor, and you're ready to begin. Before you start work, let's take a look at what Visual Studio created. Figure 2.3 shows the new project in the Solution Explorer.



Solution Explorer

- Solution 'SilverlightHello' (1 project)
  - SilverlightHello
    - Properties
      - AppManifest.xml
      - AssemblyInfo.cs
      - WMAppManifest.xml
    - References
      - Microsoft.Phone
      - Microsoft.Phone.Interop
      - mscorlib
      - system
      - System.Core
      - System.Net
      - System.Windows
      - System.Xml
    - App.xaml
      - App.xaml.cs
    - ApplicationIcon.png
    - Background.png
    - MainPage.xaml
      - MainPage.xaml.cs
    - SplashScreenImage.jpg

Figure 2.3 Files in the Solution Explorer

The project structure mirrors that of a regular Silverlight project with Properties and References folders, App.xaml, MainPage.xaml, AppManifest.xml, and AssemblyInfo.cs. Along with the references to the `Microsoft.Phone` assemblies, a few additional files are present:

- WMAppManifest.xml
- ApplicationIcon.png
- Background.png
- SplashScreenImage.jpg

The PNG image files are used by the operating system when displaying the application in the Start Experience, Application List, or Games Hub, and the splash image is shown when the Silverlight application starts up. We'll look at the image files in more depth later in the chapter.

> **NOTE** Background.png is used as the background for the start experience tile. It's not intended to be the default background of the application.

WMAppMainfest.xml contains metadata for the application, providing important details about the application to the operating system. Information in WMAppManifest.xml is also used by the Application Marketplace to validate and list an application. Visual Studio adds the WMAppManifest file to the .xap file deployment package when it builds an application. The final WMAppManifest.xml file that appears in the package downloaded to a user's phone will not necessarily contain the same information the developer specified when they built the application before submitting it to the Marketplace. During the marketplace certification, the application is examined and its manifest file is updated. A product identifier is added, the hub type or genre is set, and the security capabilities are confirmed.

Many of the settings in WMAppManifest.xml are set via the project property pages. Open the WMAppManifest.xml file and look for the `App` element, specifically the `Genre` attribute:

```
<App xmlns="" ProductID="{65438a9e-0537-451f-aaec-6ff25ca0bf85}"
    Title="Hello World" RuntimeType="Silverlight" Version="1.0.0.0"
    Genre="Apps.Normal" Author="" Description="" Publisher="">
```

The `Genre` attribute declares whether the application appears in the Application List or the Games Hub. When developing and testing on the emulator, you should leave the genre set to `Apps.Normal` since the Games Hub isn't present on the emulator. If you want to test integration with the Games Hub on a real device, you can change the setting to `Apps.Games`.

Your new Hello World project is ready to be built and deployed to the emulator or a phone. Visual Studio's Debugger is used to debug running Windows Phone applications.

2.1.1 Debugging phone projects

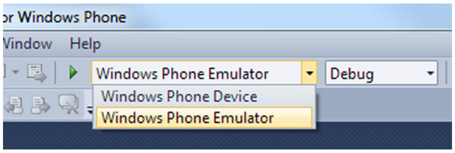


Figure 2.4 Target Deployment Device selector

Once you've built a project, you'll be able to debug it both in the emulator and on a real device. Before starting a debug session, you'll want to confirm the appropriate target is selected in the target deployment device combo box. In figure 2.4, you can see that Windows Phone Emulator is the target device, and the application will be launched in the emulator.

The first time you launch an application in the emulator, it'll take some time to boot and initialize the emulator prior to starting the application. You can also start the emulator ahead of time from the *Windows Phone SDK* folder in the Start Menu. Once the application has been launched, you'll be able to debug and interact with it in the emulator. The Windows Phone emulator can be kept running between debugging sessions.

> **TIP** An application can detect whether it's running in the emulator by checking the value of the `Microsoft.Devices.Environment.DeviceType` static property. If the value is `DeviceType.Emulator`, the application is running in the emulator.

Prior to launching an application on a real device, the phone must be plugged into the USB port and connected to your computer. The phone is considered connected when the Zune software is running. If you don't want to keep the Zune software running, you can connect the phone with the WPConnect tool. Before you can deploy and debug an application on a real phone, you must register your phone with the Developer Registration Tool. The WPConnect and Developer Registration tools were introduced in chapter 1.

When the application is being debugged, it'll automatically stop on the breakpoints you have set in the source code. You can add or remove breakpoints during the execution just like you would in any other desktop or Silverlight project. Finally, when you're done, you can stop debugging in the IDE or press the Back button on the device.

Visual Studio allows you to install or deploy the project on the device without starting a debugging session. You can do this by right-clicking on the project name in the Solution Explorer and selecting the Deploy option from the menu. The application will be copied to the device, and can be launched from the Applications List on the phone. When the application is launched, Silverlight for Windows Phone adds a few custom steps to the startup process that aren't found in Silverlight for the browser applications.

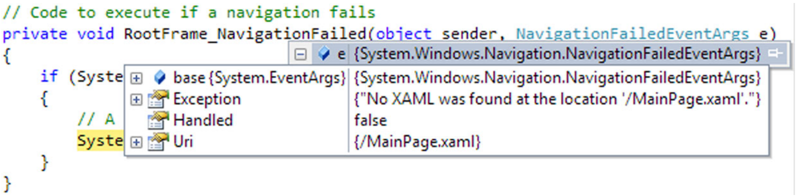
2.1.2 Application startup

Like any other Silverlight application, the entry point is a `System.Windows.Application`-derived class found in App.xaml. All phone applications are Silverlight Navigation Applications. In your phone application, the `App` class creates an instance of `PhoneApplicationFrame`, which is used as the `RootVisual`. Behind the scenes, the application host calls the `NavigationService` directly to navigate to MainPage.xaml during initial launch. When an application is reactivated, the application host navigates directly to the active page's XAML—see chapter 3 for more details on application launching and activation.

All this magic navigation is well and good; the application starts and `MainPage` is automatically loaded. What happens when you decide to rename MainPage.xaml? Since the XAML filename doesn't have to match the name of the C# class that it contains, changing the name of `MainPage` is a two-step process. Fortunately, Visual Studio's refactoring features make this a simple operation. In the Solution Explorer, right-click MainPage.xaml, choose Rename, and change the filename to HelloPage.xaml. Next, open MainPage.xaml.cs, select the `MainPage` text in the class definition, and choose Rename from the Refactor menu, specifying `HelloPage` as the new name.

When you debug the application now, the `App.RootFrame_NavigationFailed` event handler is called. Figure 2.5 shows the `NavigationFailedEventArgs` properties sent to the event handler.

Even though you renamed MainPage.xaml, the application is still configured to use MainPage.xaml as the startup URI. The startup URI is declared in the WMAppManifest.xml file as the `NavigationPage` attribute of the `Task` named "_default". Though a number of the WMAppManifest.xml settings can be set using the project property editor, the default task URI must be specified by directly editing the XML



```
// Code to execute if a navigation fails
private void RootFrame_NavigationFailed(object sender, NavigationFailedEventArgs e)
{
    if (Syste
    {
        // A
        Syste
    }
}
```

e	{System.Windows.Navigation.NavigationFailedEventArgs}
base {System.EventArgs}	{System.Windows.Navigation.NavigationFailedEventArgs}
Exception	{"No XAML was found at the location '/MainPage.xaml'."}
Handled	false
Uri	{/MainPage.xaml}

Figure 2.5 `NavigationFailedEventArgs` **properties after renaming MainPage.xaml**

file. Open WMAppManifest.xml, update the URI attribute, and then save and run the application:

```
<Tasks>
  <DefaultTask  Name="_default"
    NavigationPage="HelloPage.xaml"/>
</Tasks>
```

In this section you created a new Windows Phone Application project and examined the files created by the project template. You learned that a `PhoneApplicationFrame` is the root visual for the application and that the Silverlight runtime uses the navigation framework to load the startup page. Now that the application is running, and you've customized the name of the startup page, you'll customize the page contents.

2.2 Implementing Hello World

The project template created a default main page for your application, which you just renamed to HelloPage.xaml. In this section you're going to add a second page to the application that will display a greeting message. The second will employ a pair of `TextBlock` controls as well as a `RichTextBox`. You're also going to customize `HelloPage` by drawing a globe, as well as asking the user to input their name. First we'll take a closer look at the page created for you by the project template.

2.2.1 Customizing the startup page

The Windows Phone Application project template created the startup page with several elements meant to match the page design to the Metro style described in Microsoft's *User Experience Design Guidelines for Windows Phone*. The design guide, found on MSDN, details the expected look and feel of phone applications. The following listing shows the XAML markup added by the project template for `HelloPage`'s content.

Listing 2.1 `HelloPage`'s content as created by the project template

```
<Grid x:Name="LayoutRoot"
      Background="Transparent">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <!--TitlePanel contains the name of the application and page title-->
    <StackPanel x:Name="TitlePanel"
                Grid.Row="0"
                Margin="12,17,0,28">
        <TextBlock x:Name="ApplicationTitle"
                   Text="MY APPLICATION"
                   Style="{StaticResource PhoneTextNormalStyle}" />
        <TextBlock x:Name="PageTitle"
                   Text="page name"
                   Margin="9,-7,0,0"
                   Style="{StaticResource PhoneTextTitle1Style}" />
    </StackPanel>
```

① **LayoutRoot Grid control with two rows**

② **TitlePanel with two TextBlocks**

```
    <!--ContentPanel - place additional content here-->
    <Grid x:Name="ContentPanel"
          Grid.Row="1"
          Margin=”12,0,12,0”>
    </Grid>
</Grid>
```

❸ **ContentPanel for all other markup**

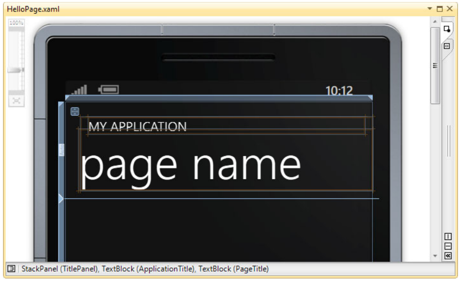

HelloPage.xaml
10:12
MY APPLICATION
page name
StackPanel (TitlePanel), TextBlock (ApplicationTitle), TextBlock (PageTitle)

Figure 2.6 HelloPage.xaml's TitlePanel in the Visual Studio Designer

The page's root layout panel is a grid control that has been split into two rows ❶. The first row contains `TitlePanel` ❷, which stacks two `TextBlock` controls for the application and page titles. The remainder of the page is allocated to the `ContentPanel` ❸. Figure 2.6 shows HelloPage.xaml as created by the project template.

Application and page titles aren't required by the design guidelines or marketplace specification but there are several rules that should be followed when they're used. The application title should be the name of the application, and should be all uppercase characters. The page title should be all lowercase characters and should describe the data or features displayed in the page. The titles shouldn't scroll or wrap; when the title doesn't fit on the screen, the text should appear truncated. If the title panel appears on the main page, it should appear on all pages to provide the user with a consistent experience.

In HelloPage.xaml, update the application title to "WINDOWS PHONE 7 IN ACTION" and the page title to "hello world":

```
<TextBlock x:Name="ApplicationTitle"
                   Text="WINDOWS PHONE 7 IN ACTION"
                   Style="{StaticResource PhoneTextNormalStyle}" />
<TextBlock x:Name="PageTitle"
                   Text="hello world"
                   Margin="9,-7,0,0"
                   Style="{StaticResource PhoneTextTitle1Style}" />
```

The title `TextBlock` controls each have their `Style` properties set to a static resource. The style resources used here won't be found anywhere in your project. They are styles injected into your application by the Silverlight framework so that your application can adhere to the user interface theme chosen by the user. Theme resources are covered in more depth in chapter 11. Theme resources are also used in the root `PhoneApplicationPage` tag to set the font and foreground color properties for the page:

```
FontFamily="{StaticResource PhoneFontFamilyNormal}"
FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
```

`PhoneApplicationPage` also has a couple of orientation properties—`Orientation` and `SupportedOrientations`. The orientation property specifies whether the current

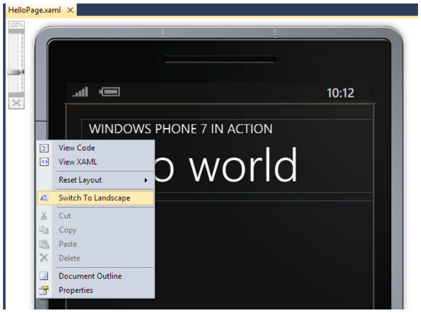


Figure 2.7 Using the context menu to switch between portrait and landscape layouts

orientation is portrait or landscape. The `SupportedOrientations` property declares which orientations are supported by the page. The visual designer supports both portrait and landscape and allows you to quickly switch between the two layouts, as shown in figure 2.7. You can read more about page orientation in chapter 11.

Windows Phone presents a status bar at the top edge of the screen in portrait layout. In landscape layout, the status bar is anchored to the edge opposite the Start button as it moves to the left or right, depending on the direction the user rotates the phone. The status bar displays the signal strength, battery, current time, and other indicators. The status bar consumes 32 pixels in portrait layout and 72 pixels in landscape layout. Screen designs should account for the space occupied by the status bar. Silverlight applications can hide the status bar with the `SystemTray.IsVisible` attached property. The project template sets this attached property to `True`. You can provide more room for your application's content and hide the status bar by setting the property's value to `False`. Before you choose to hide the status bar in your application, you should know that many users consider the status bar an essential element and dislike applications that hide it.

You're making good progress. You've gotten your hands dirty with XAML and started customizing your application. Along the way, you learned how to ensure your application fits into the system look and feel. Your next step is to add a globe and text box to the application content panel.

2.2.2 Adding application content

Remember that you want the first page of the application to draw a globe and prompt the user to enter a name. These visual elements will be added to the `ContentPanel` grid control that was created by the project template, as shown in the next listing. You'll start by dividing the `ContentPanel` into two rows, with one row using two thirds of the panel, and the remaining third allocated to the second row.

Listing 2.2 Drawing the globe

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Resources>
        <SolidColorBrush x:Key="GlobeBrush"
             Color="{StaticResource PhoneAccentColor}" />
    </Grid.Resources>
    <Grid.RowDefinitions>
        <RowDefinition Height="2*" />
        <RowDefinition Height="1*" />
    </Grid.RowDefinitions>
    <Canvas Width="200" Height="200" VerticalAlignment="Center"
```

① **GlobeBrush is static resource**

```
        Background="{StaticResource PhoneBackgroundBrush}" >
        <Ellipse Width="200" Height="200"
            Stroke="{StaticResource GlobeBrush }"
            StrokeThickness="10" />
        <Ellipse Width="100" Height="200"
            Canvas.Left="50"
            Stroke="{StaticResource GlobeBrush }"
            StrokeThickness="5" />
        <Path Data="M 100,0 100,200"
            Stroke="{StaticResource GlobeBrush }"
            StrokeThickness="5" />
        <Path Data="M 0,100 200,100"
            Stroke="{StaticResource GlobeBrush }"
            StrokeThickness="5" />
        <Path Data="M 30,40 A 100,50 0 0 0 170,40"
            Stroke="{StaticResource GlobeBrush }"
            StrokeThickness="5" />
        <Path Data="M 30,160 A 100,50 0 0 1 170,160"
            Stroke="{StaticResource GlobeBrush }"
            StrokeThickness="5" />
    </Canvas>
</Grid>
```

② **Binding to the theme brush**

③ **Binding to GlobeBrush**

The globe is drawn using Silverlight's `Ellipse` and `Path` drawing primitives. These are just two examples of the drawing primitive classes found in the `System.Windows.Shapes` namespace. The sphere of the globe and the two arced meridians are drawn with ellipses. The straight meridian and the three parallels are drawn with paths. The drawing canvas is centered in the first row of the `ContentPanel`.

> **TIP** To improve an application's performance, Microsoft recommends that complex XAML graphics be captured in a PNG or JPG and displayed with an image control.

Each of the shapes has its `Stroke` property bound ③ to a static resource you create named `GlobeBrush` ①. `GlobeBrush` has its `Color` property bound to another static resource named `PhoneAccentColor`. The canvas has its `Background` bound to a static resource named `PhoneBackgroundBrush` ②. `PhoneBackgroundBrush` and `PhoneAccentBrush` are other examples of the system theme resources that the Silverlight Framework injects into a Silverlight application. Both Expression Blend and the Visual Studio designer allow selecting system resources from their respective property windows. The Expression Blend resource menu, shown in figure 2.8, is accessed via the property editor.

The Visual Studio resource picker, shown in figure 2.9, is accessed from the Apply Resource option in a property's Advanced Options menu.

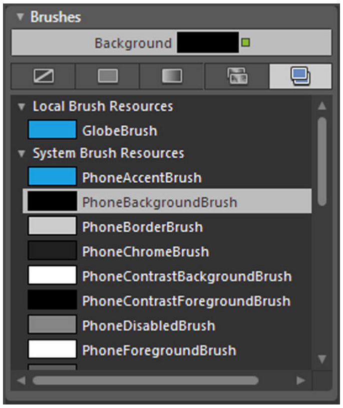
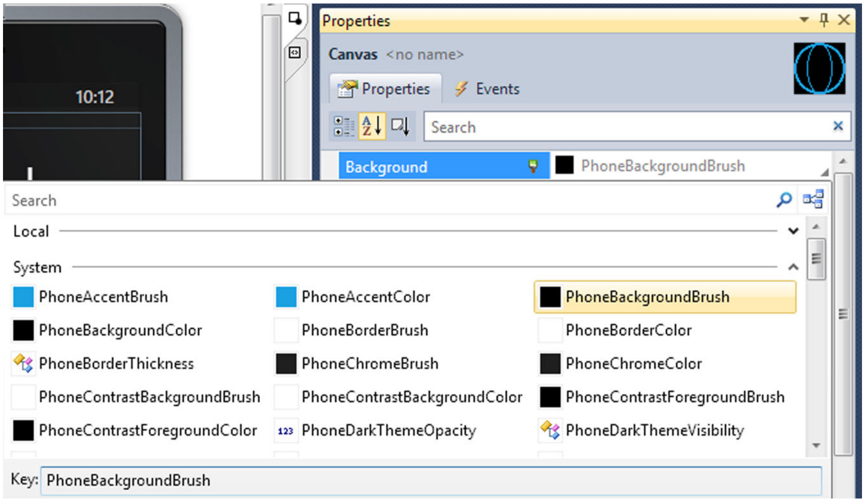

Brushes
Background
Local Brush Resources
GlobeBrush
System Brush Resources
PhoneAccentBrush
PhoneBackgroundBrush
PhoneBorderBrush
PhoneChromeBrush
PhoneContrastBackgroundBrush
PhoneContrastForegroundBrush
PhoneDisabledBrush
PhoneForegroundBrush

Figure 2.8 Expression Blend's System Brush Resources selector


10:12
Properties
Canvas <no name>
Properties
Events
Search
Search
Local
System
PhoneAccentBrush
PhoneAccentColor
PhoneBackgroundBrush
PhoneBackgroundColor
PhoneBorderBrush
PhoneBorderColor
PhoneBorderThickness
PhoneChromeBrush
PhoneChromeColor
PhoneContrastBackgroundBrush
PhoneContrastBackgroundColor
PhoneContrastForegroundBrush
PhoneContrastForegroundColor
PhoneDarkThemeOpacity
PhoneDarkThemeVisibility

Background PhoneBackgroundBrush

Key: PhoneBackgroundBrush

Figure 2.9 Visual Studio's System resource menu

In this section, you added XAML markup to draw a globe, and bound the globe elements to system brushes to enable theme support. You still need to add UI controls to implement the remaining requirement, which is to navigate to the greeting page and display the user's name. First you need to create the greetings page.

2.2.3 *Adding the greetings page*

The second page of your application will display a greeting message to the user, using the name typed into the main page. Add the new page using the Windows Phone Portrait Page item template and name the file GreetingPage.xaml. The Portrait Page item template is one of several item templates that ship with the Windows Phone Developer Tools. Table 2.2 lists the Windows Phone item templates.

Table 2.2 Windows Phone item templates

Page Template	Description
Windows Phone Portrait Page	A basic application page with title and description fields. The `Orientation` and `SupportedOrientations` properties are set to `Portrait`.
Windows Phone Landscape Page	An application page identical to a portrait page, except that the `Orientation` and `SupportedOrientations` properties are set to `Landscape`.
Windows Phone User Control	A starting point for creating reusable XAML-based controls.
Windows Phone Panorama Page	Adds an application page with `Panorama` control as its only content element.
Windows Phone Pivot Page	Adds an application page with `Pivot` control as its only content element.

The new greeting page contains controls for the application and page title. Following the same steps described for the hello page, change the application title to “WINDOWS PHONE 7 IN ACTION” and the page title to “greetings”.

The greetings page will use a couple of `TextBlocks` and a `RichTextBox` control to display the message. The XAML markup for the page’s content panel is shown in the next listing.

Listing 2.3 GreetingPage’s content

```
<Grid x:Name="ContentPanel"
      Grid.Row="1"
      Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="72" />
        <RowDefinition Height="100" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <TextBlock Margin="{StaticResource PhoneMargin}"
        Text="Hello" Style="{StaticResource PhoneTextLargeStyle}" />

    <TextBlock x:Name="helloMessage" Grid.Row="1"
        Margin="{StaticResource PhoneMargin}" Text="name goes here"
        Style="{StaticResource PhoneTextExtraLargeStyle}" />

    <RichTextBox Grid.Row="2" Margin="{StaticResource PhoneMargin}"
        VerticalAlignment="Bottom">
        <Paragraph FontSize="{StaticResource PhoneFontSizeNormal}">
            Welcome to</Paragraph>
        <Paragraph FontSize="{StaticResource PhoneFontSizeMedium}"
            Foreground="{StaticResource PhoneAccentBrush}" >
            <Italic>Windows Phone 7 in Action</Italic>
        </Paragraph>
        <Paragraph FontSize="{StaticResource PhoneFontSizeNormal}">
            Written by Massimo Perga, Timothy Binkley-Jones
            and Michael Sync.</Paragraph>
    </RichTextBox>
</Grid>
```

❶ Using PhoneMargin

❷ TextBlock for user’s name

❸ Read-only RichTextBox

You start by dividing the content panel into three rows, specifying fixed heights for the first two rows. In the first row you place a `TextBlock` containing the text “Hello”. You use the `PhoneMargin` resource ❶ to align the controls with the `TextBlocks` in the title panel. Next, you add a second `TextBlock` and give it the name `helloMessage` ❷. You’ll use this `TextBlock` to display the name of the user. Finally, you add a `RichTextBox` ❸ which you use to display formatted text. On Windows Phone, the `RichTextBox` is read-only.

You now have the two pages in your Hello World application setup and ready to go. If you run the application now, you’ll see the hello page with the nice globe. Other than look at the globe, you can’t do anything in the application. You still need to add input controls to capture the user’s name. You also need something the user can use to navigate to the greetings page. Let’s take a look at how you interact with the user.

2.3 Interacting with the user

Silverlight for Windows Phone provides most of the core user input controls that are available to Silverlight for the browser. The input controls have been modified and restyled to work in a touch-only environment and have new events that are raised when the user touches the screen. To maintain compatibility with Silverlight for the browser, the Windows Phone controls also provide mouse-related events and automatically promote touch events into mouse events. Unless you're specifically looking for touch events, you'll work with the input controls in nearly the exact same way you did when building browser applications.

There will be situations where you want to work with the touch events and gestures. Raw touch events are decomposed into *start*, *delta*, and *stop* events. *Touch gestures* combine several raw touch events into well-known gestures such as Tap, Double Tap, Hold, Pinch, Pan, and Flick. In this section you'll learn how to capture Tap and Double Tap gestures to change the color of the globe. First let's take a closer look at how the common `TextBox` control operates on Windows Phone.

2.3.1 Touch typing

The Hello World application uses a `TextBox` control for text entry and a `TextBlock` for a label. These two controls are placed inside a `StackPanel` and added to the second row of the `ContentPanel`. Since you'll need to reference the `TextBox` from code when you display the greeting message, give it the name `nameInput`:

```
<StackPanel Grid.Row="1" Margin="{StaticResource PhoneMargin}">
    <TextBlock>Enter your name:</TextBlock>
    <TextBox x:Name="nameInput" InputScope="Text"/>
</StackPanel>
```

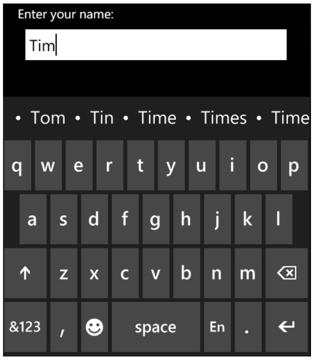
When you run the application (see figure 2.1), you'll notice that the font sizes for the two controls are different, even though you didn't specify any font information. The `TextBlock` adopts the `FontSize` of its parent containers, in this case from the page itself. Remember that the project template set the page's `FontSize` to the `PhoneFontSizeNormal` system resource.

The `TextBox` retrieves its font information from the default `TextBox` control template. The `TextBox` control template sets `FontSize` to the `PhoneFontSizeMediumLarge` system resource:

```
<Setter Property="FontFamily"
        Value="{StaticResource PhoneFontFamilyNormal}"/>
<Setter Property="FontSize"
        Value="{StaticResource PhoneFontSizeMediumLarge}"/>
```

When the user touches inside the `TextBox` the on-screen keyboard is displayed if the device doesn't have a physical keyboard. By default, the standard QWERTY keyboard is displayed as shown in figure 2.10. We recommend that you always specify an `InputScope`, even if you just use the `Text` input scope that you've used here. The `Text` input scope provides word correction features that aren't available with the default input scope. Other keyboard layouts, such as `Number` or `Url`, can be specified. `InputScopes` are covered in more depth in chapter 11.

The on-screen keyboard also exposes clipboard copy and paste operations. `TextBox` automatically supports the clipboard, and your application doesn't need to do anything special to enable clipboard operations. Developers can programmatically copy text to the system clipboard to share with other applications. Before we show you how to copy text to the clipboard, let's look at how touch gestures are supported. Your application can listen for `Tap` gestures and perform custom actions in response to gesture events.



Enter your name:

Tim

Figure 2.10 Inputting text with the on-screen keyboard

2.3.2 Touch gestures

The *User Experience Design Guidelines for Windows Phone* defines the touch gestures Tap, Double Tap, Hold, Pan, and Flick. The initial Windows Phone SDK didn't expose any gestures from Silverlight controls. The Windows Phone SDK 7.1 introduced three gesture events:

- `Tap`
- `DoubleTap`
- `Hold`

To demonstrate how touch gestures can be used in an application, you're going to change the color of the globe when it's tapped by the user. Changing the color of the globe can be accomplished by changing the color of the brush used to draw the globe's ellipse and path graphics. Remember that you bound all of the graphic elements to the static resource named `GlobeBrush`. To access the brush resource from code, you need to define a field and then initialize the field with the `SolidColorBrush` that's stored in the `ContentPanel`'s resource dictionary:

```
SolidColorBrush globeBrush;

public HelloPage()
{
    InitializeComponent();
    globeBrush = (SolidColorBrush)ContentPanel.Resources["GlobeBrush"];
}
```

Before you implement the `Tap` and `DoubleTap` event handlers, you need to add a couple of fields to enable color changes. The first is an array of colors and the second is an index of the current color:

```
Color[] colors = new Color[] { Colors.Red, Colors.Orange,
    Colors.Yellow, Colors.Green, Colors.Blue, Colors.Purple };
int colorIndex = 0;
```

Next, you hook up the `Tap` and `DoubleTap` events to the canvas panel containing the globe:

```
<Canvas Width="200" Height="200" VerticalAlignment="Center"
    Background="{StaticResource PhoneBackgroundBrush}"
    Tap="Canvas_Tap" DoubleTap="Canvas_DoubleTap">
```

In the `Tap` event handler you want to assign the `globeBrush`'s `Color` property to the next color in the `colors` array. Don't forget to check the index and reset it to the beginning of the array:

```
private void Canvas_Tap(object sender, GestureEventArgs e)
{
    colorIndex++;
    if (colorIndex >= colors.Length)
        colorIndex = 0;
    globeBrush.Color = colors[colorIndex];
}
```

In the `DoubleTap` event handler you reset the brush color to the accent color provided by the system theme. The accent color can be obtained from the application resources:

```
private void Canvas_DoubleTap(object sender, GestureEventArgs e)
{
    globeBrush.Color = (Color)App.Current.Resources["PhoneAccentColor"];
}
```

If your application requires gestures beyond tap and hold, you'll need to process the raw manipulation events raised by the Silverlight controls. The `UIElement` class exposes `ManipulationStarted`, `ManipulationDelta`, and `ManipulationCompleted` events when a user touches, moves, and releases their finger from the screen. Converting manipulation events into gestures is beyond the scope of this book.

Now that you've learned about gestures, let's discuss how to copy text to the system clipboard. Your application will copy text to the clipboard when a toolbar button is pressed.

2.3.3 Adding a toolbar button

Windows Phone provides a built-in toolbar and menu control called the *application bar*. The Visual Studio project adds sample, commented out, application bar markup when the page is created. For your Hello World application, you need to add one button to `HelloPage` and three buttons to `GreetingPage`. In HelloPage.xaml, you'll replace the sample `ApplicationBar` buttons with your own button. Start by uncommenting the `ApplicationBar` markup and removing the second button and both example menu items:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="False">
        <shell:ApplicationBarIconButton Text="say hello"
            IconUri="/Images/appbar.next.rest.png" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Next, set text for the remaining button, named `appbar_button1`, to "say hello". Finally, you need to specify an `IconUri`.

For `GreetingPage`, you also need to create an `ApplicationBar` and add buttons. You need three buttons: one labeled "ok", a second labeled "copy", and one more that's labeled "pin":

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton Text="ok"
            IconUri="/Images/appbar.check.rest.png" />
        <shell:ApplicationBarIconButton Text="copy"
            IconUri="/Images/appbar.save.rest.png" />
        <shell:ApplicationBarIconButton Text="pin"
            IconUri="/Images/appbar.next.rest.png" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

In this application, you're using a few of the icons from the Windows Phone SDK, which are installed to c:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Icons\Dark. On 64-bit Windows, the SDK is installed in c:\Program Files (x86).

Create a project folder named Images and add the appbar.next.rest.png, appbar .check.rest.png, and appbar.save.rest.png files to the folder. For each of the images, set the Build Action to Content. More information about the application bar can be found in chapter 10.

Like any other button, `ApplicationBarIconButtons` raise a `Click` event when the user presses them. You'll next register for the `Click` event on the copy button and implement the event handler. Update the button's markup to declare the event handler:

```
<shell:ApplicationBarIconButton Text="copy"
    IconUri="/Images/appbar.save.rest.png" />
    Click="copyButton_Click" />
```

Next, add the event handler in GreetingPage.xaml.cs:

```
private void copyButton_Click(object sender, EventArgs e)
{
    string message = string.Format("Hello {0}!", helloMessage.Text);
    Clipboard.SetText(message);
}
```

The event handler constructs a message by concatenating the word Hello and the text in the `helloMessage` `TextBlock`. The greeting message is then copied to the `Clipboard` and is ready to be pasted into some other application.

In this section you learned how to receive typed text from the user, respond to touch gestures, and use the system application bar to display a toolbar buttons. You implemented a click handler for one of the toolbar buttons, but the other two buttons don't perform any work. The unimplemented buttons will be used to navigate between the two pages, which we cover in the next section.

The Model-View-ViewModel pattern

Many, but not all, Silverlight developers use the Model-View-ViewModel pattern (MVVM) to separate user interface markup and logic from application logic. The separation of UI and application logic promoted by MVVM is made possible with Silverlight's data binding, value converter, and commanding features. Input and `TextBlock` controls are bound to model objects, which often implement the `INotifyPropertyChanged` interface. Values are converted to strings using converter classes that implement `IValueConverter`. Click event handlers are eschewed in favor of command objects implementing the `ICommand` interface.

Though MVVM separates UI and business logic, it introduces complexity. We've intentionally avoided using the complexity of the MVVM pattern in the sample applications in the book. We've also avoided binding trivial properties such as messages displayed in a `TextBlock`, and have placed a great deal of our application logic in the page code behind. MVVM is a great pattern that's well suited for XAML applications but one criticism of MVVM is that it's overkill for simple applications.

This isn't a book about Silverlight, but about Windows Phone. The bits of Silverlight we use in the sample applications are intended to highlight the features of the Windows Phone SDK that aren't available to browser-based Silverlight applications.

2.4 Page navigation

A phone application is a modified version of a Silverlight Navigation Application. Silverlight Navigation Applications are composed of a navigation frame and one or more pages that interact with the `NavigationService`. The `NavigationService` interacts with the operating system to maintain a journal or history of pages visited by the user. In this section, you're going to add navigation to the Hello World application.

2.4.1 Navigating to another page

Page navigation is the process that takes the user from one page to another. One example is when the user presses a button to open a new page, and then after completing some work, presses another button to come back to the main page. Navigation is managed by the `NavigationService` class. The `NavigationService.Navigate` method is called to move to a new page. When `Navigate` is called, the current page is placed on the navigation stack, and a new instance of the target page is generated. The `NavigationService.GoBack` method removes the current page and restores the previous page that's on the navigation stack.

You'll now add page navigation to your Hello World application. Starting in HelloPage.xaml add a click event handler to the "say hello" button:

```
<shell:ApplicationBarIconButton Text="say hello"
    IconUri="/Images/appbar.next.rest.png"
    Click="navigateForwardButton_Click" />
```

You want to navigate to `GreetingPage` when the button is pressed, so you need to add code to the click handler:

```
private void navigateForwardButton_Click(object sender, RoutedEventArgs e)
{
    this.NavigationService.Navigate(
        new Uri("/GreetingPage.xaml", UriKind.Relative));
}
```

You access the `NavigationService` via the `PhoneApplicationPage`'s `NavigationService` property. The `Navigate` method accepts an `Uri`, which in this case is the name of the file containing the page you wish to load. You construct the `Uri` using `UriKind.Relative`, as it's part of the same XAP file.

Now you want to reopen GreetingPage.xaml and generate the click event handler for the OK application bar button:

```
<shell:ApplicationBarIconButton Text="ok"
    IconUri="/Images/appbar.check.rest.png"
    Click="navigateBackButton_Click" />
```

You implement the handler by calling the `GoBack` method:

```
private void navigateBackButton_Click(object sender, EventArgs e)
{
    this.NavigationService.GoBack();
}
```

Now press F5, or select the Debug->Start Debugging menu option, and debug the application. You've just linked your two pages using only two lines of code. Press the hello button and see the second page appear. When you press the OK button, the main page appears again.

It's worth noting that you use the `GoBack` method instead of the `Navigate` method to return to MainPage.xaml. When you call `GoBack` the current page is removed from the page stack. If you'd used `Navigate`, a new page would've been added on top of the page stack. Depending on the scenario you want to achieve, you can choose the approach more appropriate for your application, but you must be aware of the consequences. Both the cases are illustrated in Figure 2.11.

Let's examine the two scenarios presented in figure 2.11. In the top sequence the navigation uses `GoBack` to return to `HelloPage`, so the page stack is reduced. In the sequence on the bottom, the navigation uses `Navigate` to navigate to `HelloPage`, and a new page is added on top of the page stack and made visible.

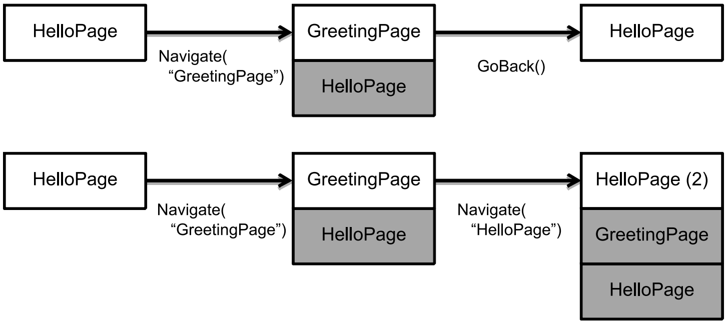


Figure 2.11 The navigation page stack resulting from GoBack (top) and Navigate (bottom) method calls. The white boxes represent the visible page, whereas the shaded boxes are the pages in the background.

Your Hello World application now moves from one page to another, and the greeting page starts up as expected. How do you get the user-entered name from the hello page to the greeting page? The Silverlight Navigation Framework provides features to enable passing data into a newly launched page.

2.4.2 Passing parameters between pages

In the previous example you concentrated on navigation between pages, but didn't pass any information to the greeting page. In theory, pages should be as self-contained as possible in order to maintain isolation between the pages, but it can be useful to pass parameters when navigating. You could choose to use some form of global data or data cached in the `App` class instead of passing data, but you should consider passing parameters in the `Uri` much as you would pass data to a constructor. As you'll learn later in the chapter, the operating system can call your page directly without ever constructing an instance of your main page.

In your sample main page, the user enters a name into a text box control named `nameInput` whose `Text` property will be used as a parameter passed to `GreetingPage`. `GreetingPage` will set the text block having name `helloMessage` with the parameter passed by `HelloPage`. Two changes are required in your code to pass a parameter—`HelloPage` must pass the parameter value to `GreetingPage` via the navigation `Uri` and `GreetingPage` must extract the parameter value from the query string.

Earlier you just created a URI with the hard-coded name of the greeting page in the `navigateForwardButton_Click` method. You could choose to hard-code the parameters as well, but now you have more magic strings in your code. What if you change the name of the greeting page, or change the name of the parameters passed in the query string? You'll next move `Uri` construction code into a static method of the `GreetingPage` class. Modify the `Uri` to pass a parameter in the same manner that you would if you were adding fields to a standard HTTP query string:

```
public static Uri BuildNavigationUri(string name)
{
    return new Uri("/GreetingPage.xaml?name=" + name, UriKind.Relative);
}
```

Update the `navigateForwardButton_Click` method in HelloPage.xaml.cs to call the new factory method, passing along the name entered by the user. The parameter value is obtained from the `nameInput` control:

```
NavigationService.Navigate(
     GreetingPage.BuildNavigationUri(nameInput.Text));
```

The data passed via the navigation `Uri` can be retrieved from the target page's `NavigationContext` property. The `NavigationContext` class has a single property named `QueryString`, which is an `IDictionary<string, string>` mapping parameter names to values.

You'll use the `NavigationContext` in the code behind for GreetingPage.xaml. The appropriate time to access the query string is after the page navigation has completed.

The navigation framework calls the `PhoneApplicationPage.OnNavigatedTo` virtual method when navigation is complete. In the sample application, you override `OnNavigatedTo` and obtain the parameter value by using the string `"name"` as a key into the `QueryString`. You set the returned value into `Text` property of `helloMessage`:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    helloMessage.Text =
        this.NavigationContext.QueryString["name"];
}
```

`OnNavigatedTo` is one of the virtual methods defined by `PhoneApplicationPage` that are called when navigation events occurs. `OnNavigatedFrom` and `OnNavigatingFrom` are two other methods you can use to determine when the current page is changing.

In this section we've shown how you can navigate between pages from your software. Other activities, such as the user pressing the hardware Back key, can cause navigation changes in your application. Every Windows Phone is equipped with a hardware Back key. Its effect in an application is equivalent to calling the `NavigationService.GoBack` method.

2.4.3 Changing the Back key behavior

When the Back key is pressed, the navigation framework automatically performs a `GoBack` operation. The Back key behavior can be interrupted, for instance, to avoid moving off a page that has unsaved changes. To interrupt the automatic `GoBack`, the `PhoneApplicationPage` class provides an event named `BackKeyPress`. You'll see this event in action by wiring it to an event handler in your `GreetingPage` class. You first need to edit GreetingPage.xaml by adding an attribute to the `PhoneApplicationPage` tag:

```
BackKeyPress="Page_BackKeyPress"
```

Next, you add the event handler to GreetingPage.xaml.cs. This example prompts the user with a confirmation message:

```
private void Page_BackKeyPress(object sender, CancelEventArgs e)
{
    MessageBoxResult result = MessageBox.Show(
        "Press OK to return to the previous page.",
        "WP7 in Action", MessageBoxButton.OKCancel);
    if (result == MessageBoxResult.Cancel)
        e.Cancel = true;
}
```

If the user presses the Cancel button in the message box, you set the `CancelEventArgs.Cancel` property to true. This cancels the default behavior of the Back key. Failing to add this statement or setting `e.Cancel` to false would have maintained the default behavior, which is to move to the previous page or to terminate the application if no other pages are in the page stack.

> **NOTE** The *Application Certification Requirements for Windows Phone* details appropriate application behavior when working with the back key. Specifically, when the Back button is pressed while the main page is visible, the application must exit.

Navigation relies on URI and query strings to navigate to specific locations within an application. Navigation strings can also be used to allow a user to launch to a specific location within your application with application tiles

2.4.4 Navigating with tiles

Windows Phone users can pin an application's tile to the start screen. Tiles are large icons that display a background image and a title. We show you how to customize the main application tile in the next section. Starting with Windows Phone SDK 7.5, applications can also create secondary tiles that will navigate directly to a specific location in an application. Figure 2.12 shows the application and secondary tiles for your Hello World application.

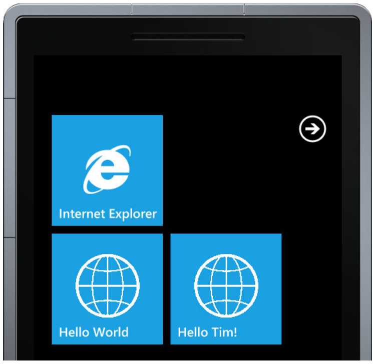


Figure 2.12 Application and secondary tiles for Hello World

When the user clicks the application tile for Hello World, the application is launched and the `NavigationService` is called with the URL `/HelloPage.xaml`. When the user clicks the secondary tile, the application is launched and the application host passes the URL associated with the secondary tile to the `NavigationService`. When you built the `GreetingPage`, you added a pin button to the application bar. Add a click handler to the pin button and implement code to create a secondary tile:

```
private void pinButton_Click(object sender, EventArgs e)
{
    StandardTileData tileData = new StandardTileData
    {
        BackgroundImage = new Uri("Background.png", UriKind.Relative),
        Title = string.Format("Hello {0}!", helloMessage.Text),
    };
    ShellTile.Create(BuildNavigationUri(helloMessage.Text), tileData);
}
```

The `StandardTileData` class has several properties that describe the tile. In your application you only use the `BackgroundImage` and the `Title` properties. You set the `BackgroundImage` property to use Background.png, the image specified in the project properties for your main tile. You set the `Title` property to be the greeting message. The `Create` method of the `ShellTile` class is used to create the new tile. You specify the `Url` to the greeting page, passing the same parameters that are specified by HelloPage. When `ShellTile.Create` is called, the application exits and the start screen is launched, showing the new secondary tile to the user.

Tiles have several other features that include flip side background images, content, and counters. Tiles can be dynamically updated or deleted by application code. These tile features are covered in chapter 9.

In this section you implemented the final requirement for your Hello World application—navigating to a second page and displaying a greeting to the user. You learned about the `NavigationService` and how to use query string parameters to pass data between pages. Finally, you learned how to use tiles to navigate directly to the greeting page from the start screen. Now you'll add some polish and customize the start-up experience with your own splash screen and other artwork.

2.5 Application artwork

The Windows Phone operating system expects your Silverlight application to provide a few different artwork files which it uses to represent your application to the user. Depending on how your application is built and configured, your application artwork can be displayed in the Start Screen, the Application List, the Games Hub, and the Music + Videos Hub. The Silverlight Framework also looks for a splash screen image when launching your application. In this section we discuss how to update or replace the artwork created by the project templates. We also discuss the image formats and sizes that are expected by Windows Phone. Let's begin with the splash screen.

2.5.1 Customizing the splash screen

When a Silverlight application is loaded, the application framework briefly displays a splash screen while constructing and navigating to the first page. If the page's constructor or the `OnNavigatedTo` methods perform lengthy operations, the splash screen will remain visible until the work is completed.

> **NOTE** The *Application Certification Requirements for Windows Phone* recommends that you only provide a splash screen image when your application takes longer than 1 second to display the first page. The certification requirements also require that the first page be shown in less than five seconds.

The splash screen displayed by the framework is a static image, and can't be updated or replaced during runtime. The image used by the framework comes from the file named SplashScreenImage.jpg. The default 480 x 800 pixel image created by project templates is a dark gray background with a clock face. A custom splash screen image can be used by simply overwriting SplashScreenImage.jpg with another file of the same name.

There are two other image files that are created by the project template: the tile image and the application icon.

2.5.2 Customizing tile images and application icons

The tile image is used by the operating system when an application or game is pinned to the start screen. Application icons are used by the operating system when an application

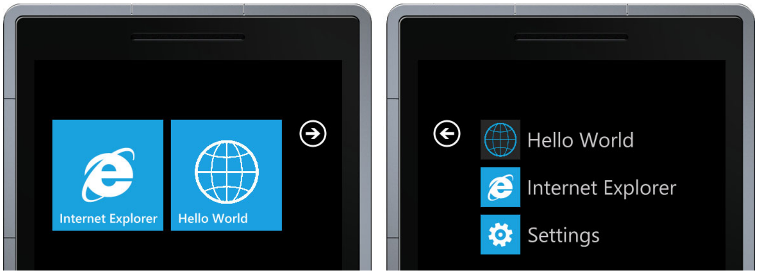


Figure 2.13 Custom images used in the start experience and the application list

appears in the application list. Figure 2.13 shows the start screen and application list custom globe images.

The Visual Studio project templates create default tile background images named Background.png for Silverlight applications. Silverlight projects are created with an application icon named ApplicationIcon.png. You can replace the default tile background image following these steps:

- Create a new 173 x 173 pixel PNG file.
- The image should have a 12-pixel margin on all edges.
- The image should reserve a 37 x 37 pixel area inside the top-right margin for tile notifications. (Tile notifications are covered in chapter 9.)
- Add the new image to the root of the project and specify the Content build action.
- In the Application tab of the project properties, select the new image in the Background image field.

The tile image is displayed in the Start Experience with the tile title. You can set a custom tile title with the Tile Title field in the Application tab of the project properties. An application can dynamically update its tile background image, which is covered in more depth in chapter 9. Applications implementing notifications commonly update the tile background image.

The tile images for the native phone application use the system theme's accent color as a background color. When the user changes the accent color, the tile background is updated to match. You can also design your tile images to use the theme's accent color by using transparency in your PNG file. Transparent pixels in the tile image will allow the accent color to show through.

You can replace the default application icon following these steps:

- Create a new 62 x 62 pixel PNG file for an application. If your application will be displayed in the Games Hub, the image should be 173 x 173 pixels.
- Add the new image to the root of the project and specify the Content build action.
- Open the Application tab of the project properties and select the new image in the Deployment Icon field.

The Games Hub expects the application icon/game thumbnail to be 173 x 173 pixels. The Application List expects the icon to be 62 x 62 pixels. ApplicationIcon.png is created by the project template as 62 x 62 pixel files. You should replace these files with larger images if your Silverlight application will appear in the Games Hub. Customizing the tile and application images improves an application's integration into the overall Windows Phone experience.

2.6 Try before you buy

Before committing hard-earned money to a purchase, many users like to use a demonstration or trial version of an application, and the Windows Application Marketplace provides support for limited trials. Applications use the `IsTrial` method of the `LicenseInformation` class to determine if the application is running under a trial license:

```
LicenseInformation licenseInfo = new LicenseInformation();
if (licenseInfo.IsTrial())
{
    // implement trial mode logic here...
}
```

The manner in which the trial mode is implemented is up to the discretion of the developer. Trial mode applications can be restricted to a certain number of days, a certain number of launches, have a restricted feature set, or some other limitation.

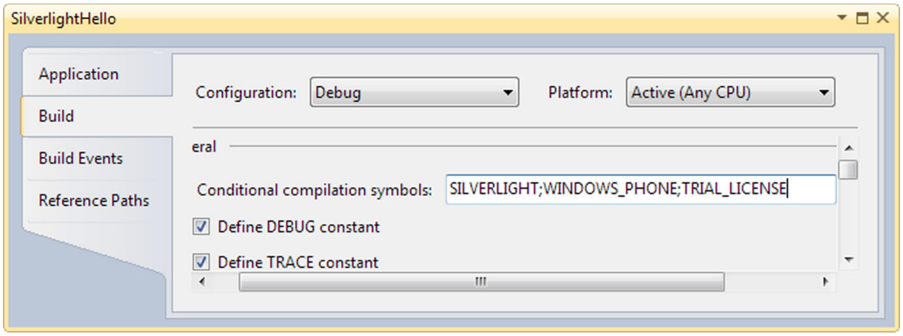
Applications running in trial mode should provide a "buy me" link to purchase the application from the Application Marketplace. The marketplace link can be implemented using the `MarketplaceDetailTask`, which is covered in section 4.2.4. Applications should re-check the trial when first launched, or when reactivated from a dormant or tombstoned state, as the user may have purchased the application while it was inactive. Checking the trial status can be time consuming and shouldn't be performed in a tight loop.

Developers should always test their applications in both trial and unlimited modes. Testing can be problematic as the `IsTrial` method always returns false when running in the emulator. Conditional compilation techniques can be used to test trial licensing:

```
    LicenseInformation licenseInfo = new LicenseInformation();
#if TRIAL_LICENSE
    bool isInTrialMode = true;
#else
    bool isInTrialMode = licenseInfo.IsTrial();
#endif
    if (isInTrialMode)
    {
        // implement trial mode logic here...
    }
```

To turn on trial licensing, all you need to do is add a conditional compilation symbol as shown in figure 2.14.

The application marketplace's trial licensing makes it easy to provide potential customers a preview of your application or game, without the need to build multiple



Configuration: Debug

Platform: Active (Any CPU)

Conditional compilation symbols: SILVERLIGHT;WINDOWS_PHONE;TRIAL_LICENSE

☑ Define DEBUG constant

☑ Define TRACE constant

Figure 2.14 Setting a conditional compilation symbol

versions of your project. Trial licensing eliminates the need to maintain and publish a separate free or light version of your product.

2.7 Summary

The Windows Phone Developer Tools help you build many different kinds of Silverlight applications. Project templates are provided for simple projects and class libraries as well as list, pivot, panorama, and Silverlight with XNA style projects. Chapter 10 covers panorama and pivot applications, whereas Silverlight with XNA applications are covered in chapter 14.

The Silverlight framework makes it easy to align your application with the system theme and style. The framework injects resources into applications so they can match the system theme (light versus dark, accent color) and the look and feel (fonts, colors, sizes). The visual designers and property editors in Expression Blend and Visual Studio expose theme resources.

Silverlight has been extended for Windows Phone with components built specifically for the platform. New navigation frame, page control, and application bar components are just a few of the additions. Other existing Silverlight controls have been modified to work on the phone. The chapters in part 3 of this book look at these new and modified components.

Finally you learned some of the procedures for integrating with the phone operating system. Live tiles and application icons can be customized to make your application stand out in the quick start, application list, and Game Hub. System capabilities must be declared in order to use many of the core phone APIs. The phone APIs provide access to the native applications, services, sensors, and media features of the phone. In part 2 of this book you read about how to use the phone APIs. In the next chapter you learn how Windows Phone implements application multitasking and how to design application-to-lifecycle events. We also look at how to create and run background agents to perform work while other applications run in the foreground.

Part 2

Core Windows Phone

Now that you understand the Windows Phone platform and how to use Visual Studio and the SDK tools, it's time to learn the low-level details on how build mobile applications. Part 2 of this book will introduce concepts that are brand-new to Windows Phone, as well as concepts that have been adapted to operate within the phone's limitations.

We start in chapter 3 with a discussion of Fast Application Switching, Microsoft's name for the battery saving technology that allows a dormant application to be quickly restored when a user switches from a foreground application to a background application. Chapter 4 will show you how to use launchers and choosers to interact with built-in applications such as the phone dialer, email, and the People Hub, while chapter 5 explains how to store your application data.

Chapters 6 and 8 show you how to read data from the phone's hardware including the camera, accelerometer, compass, and gyroscope, while chapter 7 discusses how to integrate your app with the Pictures and Music + Video Hubs. Part 2 wraps up in chapter 9 with a discussion of networking topics such as using TCP sockets and push notifications. Push notifications provide the ability for an external application or web service to send messages and updates to particular Windows Phone devices.

3

Fast application switching and scheduled actions

This chapter covers

- Fast application switching
- Responding to lifetime events
- Scheduling reminders
- Executing tasks with a background agent

Just like any other operating system, an application on the Windows Phone starts up, runs for a while, and in the normal course of things, the application eventually exits. In other multitasking operating systems, an application can be moved to the background when the user switches applications. While in the background, the application will continue to run in lower-priority time-slices. The Windows Phone OS is a multitasking operating system, but puts limitations on background operations. When a user switches applications, the running application is paused, and system resources are disconnected from the process so they can be freed up for foreground applications. A dormant application might eventually be terminated if the operating system needs to allocate additional resource to the foreground application.

Applications can't run in the background, but Windows Phone offers several options to the developer to build applications that require multitasking features.

Fast application switching provides dormant applications the tools for quickly returning to full operation, giving the user the impression that the application continued to run. Background agents are the mechanisms applications use to perform tasks even when the application isn't running. Alarms and reminders can be used to notify a user about important tasks, and allow the user to easily restart the application. Background tasks, alarms, and reminders are all different forms of a *scheduled action*, and are managed with the `ScheduledActionService`.

The multitasking limitations imposed on applications may seem severe to developers looking to build the next killer mobile application. Microsoft has imposed these limitations to ensure that the user has the best overall experience possible. Background tasks aren't allowed to affect or slow down foreground application, nor are they allowed to perform tasks that will quickly drain the phone's battery.

In this chapter you'll create two different sample applications. The first one, which you'll name Lifetime, demonstrates how to build applications that support fast application switching. The second application, which you'll name ScheduledActions, uses the `ScheduledActionService` to schedule reminders and periodic tasks, and implements an example background agent.

3.1 Fast application switching

Fast application switching is the term coined by Microsoft and the Windows Phone team to describe the process that pauses a running application when the user switches to another application. The paused application usually remains in memory, but can't execute any code. The application will resume running once the user switches back to the application, making it the foreground application once again.

An application may be paused via one of many scenarios:

- The user presses the Windows or Search button.
- The user chooses to reply to a newly received text message.
- The user chooses to responds to a newly received toast notification.
- The user presses the Camera button.
- The user locks the phone, or the idle timeout expires.
- The application shows a launcher or chooser.

When the user backs out of the application, it's shut down normally. In all other situations, the user can restart your application with the Back button or by selecting your application with the Task Switcher UI. Upon restart, your application should return to the state it was in before being paused, giving the user the impression that the application continued to run.

Your application can also be obscured. Obscuration occurs when the operating system covers part of your application in favor of another application's UI. An application may be obscured when

- The user receives a new phone call.
- The user returns to a phone call that was active before an application started.

- The user receives a new toast notification.
- The lock screen is enabled, and the application has changed idle detection mode.
- An alarm or reminder is triggered.
- An application uses the `PhoneCallTask` launcher.

Pausing and obscuration interfere with the normal flow of an application. Your application should be designed to react to interruptions by appropriately responding to the application life-time events.

3.1.1 Understanding lifetime events

The Windows Phone class libraries expose operating system lifetime events, with several events defined on a few different classes. Table 3.1 describes each of the lifetime events and where the event is implemented in the framework.

Table 3.1 Application lifetime events

Event	Purpose	Implementation
`Launching`	Notifies the application that it's being started from scratch	`PhoneApplicationService .Launching`
`Closing`	Notifies the application that it's being terminated as part of a normal application exit	`PhoneApplicationService .Closing`
`Activated`	Notifies the application that it's being resumed from a paused, dormant, or tombstoned state	`PhoneApplicationService .Activated`
`Deactivated`	Notifies the application that it's being paused to allow another process to execute	`PhoneApplicationService .Deactivated`
`Obscured`	Notifies the application that it's losing focus in favor of an overlay	`PhoneApplicationFrame .Obscured`
`Unobscured`	Notifies the application that it's regaining focus	`PhoneApplicationFrame .Unobscured`

An application begins life when the user taps the application's icon in the Application List. The application host constructs an instance of the `Application` class, raises the `Launching` event, and then constructs and navigates to the default navigation page. At this point, the application is running, solving user problems and generally making life easier. In the normal course of activity, the user presses the Back button to exit the application and then the operating system navigates away from the application and raises the `Closing` event. The launching-running-closing workflow can be seen in figure 3.1 as the straight-line path between start and end.

While running, an application will likely receive other lifetime events. When the `Obscured` event is raised, the application transitions to an obscured state. The application remains running but its user interface is partially or completely hidden. When

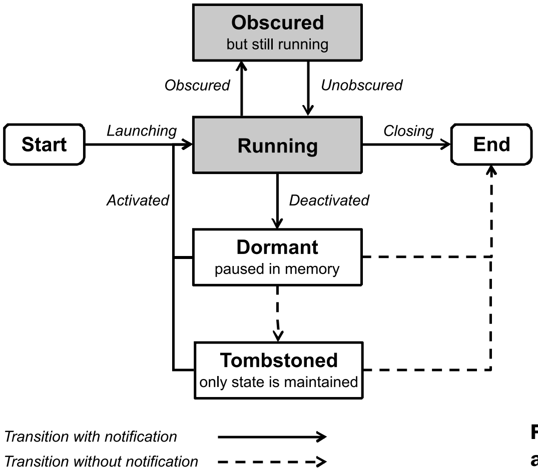

Obscured
but still running
Obscured
Unobscured
Start
Launching
Running
Closing
End
Activated
Deactivated
Dormant
paused in memory
Tombstoned
only state is maintained
Transition with notification
Transition without notification

Figure 3.1 Application life-time states and the actions that trigger state transition

the user interface is uncovered, the `Unobscured` event is raised and the application returns to the normal running state.

Deactivation is triggered when the operating system pauses an application in order to execute another process. The `Deactivated` event is raised to notify the application that it's transitioning to the dormant state. In the dormant state, the application remains in memory with all processing and threads stopped. The dormant application remains in memory to facilitate a fast restart when the user switches between applications. When the user returns to the application, the `Activated` event is raised and the application returns to the normal running state.

A dormant application might also transition to a tombstoned state. When an application is tombstoned, no notification is given, the application is removed from memory, and almost all of its resources are freed up for other applications to use. The only application resources that are maintained in memory are those items that have been stored in one of two state dictionaries provided by the Windows Phone framework specifically for tombstoning scenarios. When the user navigates back to the application, the application host constructs a new instance of the `Application` class, raises the `Activated` event, and then constructs and navigates to the previously focused page. It's the application's responsibility to save application data to the `State` dictionaries and to read and restore data after reactivation.

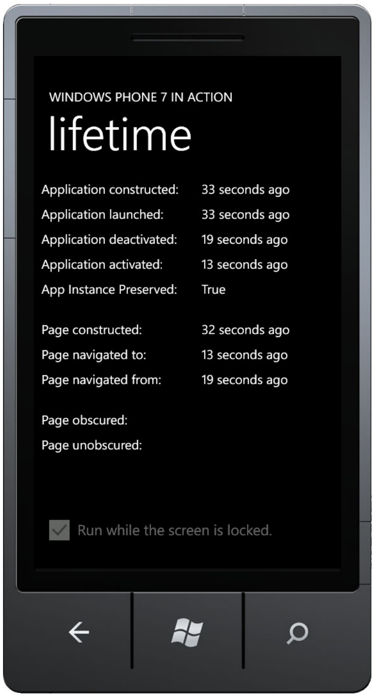
When an application is dormant or tombstoned, the operating system may decide to terminate the application and end its life. When terminated, the application and any saved state are removed from memory. An application isn't notified when transitioned to the end state.

In the first half of this chapter you build an application that handles each of the lifetime events. You start your sample application in the usual way—by creating a new project.

3.1.2 Creating the Lifetime sample application

The Lifetime sample application will handle each of the lifetime events. The sample code will demonstrate how to distinguish between returning from a dormant state and a tombstoned state. You'll learn how to store application data in the `State` dictionaries, and how to recover the data when returning from a tombstoned state. Finally, we show you how to design an application capable of running when the phone is locked.

The Lifetime application records the time when each of the lifetime events are raised, displaying the times on the screen. Moving beyond the lifetime events, we explore other interesting moments in the application lifetime such as construction and navigation. Figure 3.2 is a screenshot of the Lifetime sample application.



WINDOWS PHONE 7 IN ACTION

lifetime

Application constructed:	33 seconds ago
Application launched:	33 seconds ago
Application deactivated:	19 seconds ago
Application activated:	13 seconds ago
App Instance Preserved:	True
Page constructed:	32 seconds ago
Page navigated to:	13 seconds ago
Page navigated from:	19 seconds ago
Page obscured:	
Page unobscured:	

☑ Run while the screen is locked.

Figure 3.2 The Lifetime sample application

Create a new project using the Windows Phone Application template and name the project Lifetime. The Visual Studio Silverlight Windows Phone Application project template automatically creates lifetime event handlers for the `PhoneApplicationService` events in App.Xaml.cs. Events handlers aren't created automatically for the `PhoneApplicationFrame` events. A `PhoneApplicationService` instance is declared in App.xaml, and the `Launching`, `Closing`, `Activated`, and `Deactivated` events are wired up to methods generated in App.xaml.cs. The following listing shows the App.xaml markup and the corresponding event handlers in App.xaml.cs.

Listing 3.1 Generated application life-time event handlers

```
<Application.ApplicationLifetimeObjects>
    <shell:PhoneApplicationService
        Launching="Application_Launching"
        Closing="Application_Closing"
        Activated="Application_Activated"
        Deactivated="Application_Deactivated"/>
</Application.ApplicationLifetimeObjects>

private void Application_Launching(object sender, LaunchingEventArgs e){}
private void Application_Activated(object sender, ActivatedEventArgs e){}
private void Application_Deactivated(object sender,
    DeactivatedEventArgs e){}
private void Application_Closing(object sender, ClosingEventArgs e){}
```

An application isn't considered correct by the Windows Phone Marketplace certification process if `PhoneApplicationService` events aren't handled. The handlers must be both declared in the XAML and implemented in the code behind, at least as generated by the Visual Studio wizard.

The main page for the Lifetime sample application displays how many seconds ago a particular lifetime event occurred. The sample application will record the occurrence time of each of the lifetime events, along with a few other interesting events. The sample application displays these times as the number of seconds since the event was raised. The information for each event is displayed with two `TextBlock`s, the first displaying a label and the second displaying the number of seconds since the event. To organize all of the controls, you'll divide the `ContentPanel` into several rows and columns, as shown in the next listing.

Listing 3.2 Defining the `ContentPanel`'s layout

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="39" />
...
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
</Grid>
```

① **Declare 11 rows**

② **Declare 2 columns**

First you divide the `Grid` into 11 rows ① and two columns ②. We've omitted the declaration of most of the rows for the sake of space. Whereas most of the rows and the first column are assigned a concrete height or width, the last row and last column fill the remainder of the page's available space.

As you work through the chapter, you'll add several `TextBlock` controls to MainPage.xaml. You'll add the first few controls in the next section when you display the number of seconds elapsed since the `MainPage` was constructed.

3.2 Launching the application

You learned about application startup in chapter 2. During normal application startup, an instance of `App` and an instance of `MainPage` are constructed and the application host calls the `NavigationService` directly to navigate to MainPage.xaml. In addition to constructing these two objects, the `PhoneApplicationService` raises the `Launching` event. In this section we look at how to detect when an application is launched.

3.2.1 Construction

In the sample application, you capture when the `App` and `MainPage` instances are constructed and show the times in the user interface. The construction times are stored in properties in each of the two classes. You start your implementation updating the class in App.xaml.cs to define and assign a property named `AppConstructedTime`. The property is defined as an automatic property with a private setter. The `AppConstructedTime` property is assigned to the current time in the `App` class constructor:

```
public DateTime AppConstructedTime { get; private set; }

public App()
{
    AppConstructedTime = DateTime.Now;
    ...
}
```

You add a similar property to the `MainPage` class. In addition to recording the construction time, you also record when the `OnNavigatedTo` method is called. These two times are stored in two fields named `pageConstructedTime` and `navigatedToTime`:

```
DateTime pageConstructedTime;
DateTime navigatedToTime;
```

Now update the `MainPage` class constructor to assign the current time to the `pageConstructedTime` field:

```
public MainPage()
{
    InitializeComponent();
    pageConstructedTime = DateTime.Now;
}
```

The `navigatedToTime` field is assigned the current time in the overridden `OnNavigatedTo` method:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    navigatedToTime = DateTime.Now;
    UpdateUserInterface();
    base.OnNavigatedTo(e);
}
```

In the `OnNavigatedTo` method you also added a call to a method named `UpdateUserInterface`. The `UpdateUserInterface` method is responsible for updating the user interface with the time values stored in the `App` properties and the `MainPage` fields. `UpdateUserInterface` sets the time values into associated `TextBlock` controls. Before you implement the `UpdateUserInterface` method, you need to add a few controls to MainPage.xaml, shown in the next listing.

Listing 3.3 UI controls showing construction times

```
<TextBlock Grid.Row="0" Text="Application constructed:" />
<TextBlock x:Name="appConstructed" Grid.Row="0" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />

<TextBlock Grid.Row="5" Text="Page constructed:" />
<TextBlock x:Name="pageConstructed" Grid.Row="5" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />
```

```
<TextBlock Grid.Row="6" Text="Page navigated to:" />
<TextBlock x:Name="navigatedTo" Grid.Row="6" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />
```

In this code snippet you've added six `TextBlock` controls. Three of the `TextBlocks` are used as labels. The other three display the relevant elapsed time values and are named `appConstructed`, `pageConstructed`, and `navigatedTo`. These three controls are used by the `UpdateUserInterface` method:

```
public void UpdateUserInterface()
{
  DateTime now = DateTime.Now;
  pageConstructed.DataContext = (now - pageConstructedTime).TotalSeconds;
  navigatedTo.DataContext = (now - navigatedToTime).TotalSeconds;

  var app = (App)Application.Current;
  appConstructed.DataContext = (now - app.AppConstructedTime).TotalSeconds;
}
```

The `DataContext` properties of the `TextBlock` controls are assigned values calculated by the number of seconds elapsed between the current time and the time stored in the related variables in the `MainPage` and `App` classes.

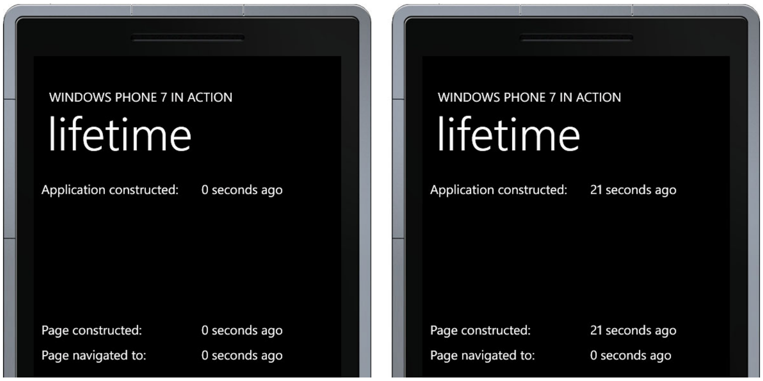
Run the application and examine the values displayed to the screen. On first run, the screen should display construction and navigation times of one or zero seconds ago. The left side of figure 3.3 shows the initial construction times. Now press the Start button and launch another application. Using the Task Switcher or the Back button, switch back to the running instance of the sample application.

> **NOTE** The Task Switcher is launched in the emulator by pressing and holding the `F1` key on your computer's keyboard. If you're running the application on a real device, press and hold the Back button until the Task Switcher is shown.

Windows Phone maintained the application process in memory, and the `App` and `MainPage` classes don't need to be recreated when the application is restored. Once the sample application is restored and running, the screen should display construction times of several seconds ago, depending on how long you took to switch back to the application. The navigation time should be zero again, since `OnNavigatedTo` is called when returning to the application. The right side of figure 3.3 shows the user interface after the restart.

In this section you added code to track when the `App` and `MainPage` classes are constructed. The classes are constructed when the application is first launched, and aren't necessarily constructed when the application returns from dormancy. Later in the chapter we'll show you situations where the operating system will tombstone an application, and the `App` and `MainPage` classes are constructed again when the application is reactivated.

Fortunately, the Windows Phone framework provides the lifetime events to determine when an application is created during initial launch, and when the application classes are recreated when the application is reactivated.



WINDOWS PHONE 7 IN ACTION

lifetime

Application constructed: 0 seconds ago

Page constructed: 0 seconds ago

Page navigated to: 0 seconds ago

WINDOWS PHONE 7 IN ACTION

lifetime

Application constructed: 21 seconds ago

Page constructed: 21 seconds ago

Page navigated to: 0 seconds ago

Figure 3.3 Lifetime application after initial construction (left) and restart (right)

3.2.2 First-time initialization

When a user taps an application's icon on the phone's Application List, a new instance of the application is launched. If an existing instance of the application is currently dormant, it'll be removed in favor of the new instance. Other triggers that create a new instance of an application include the user tapping an application or secondary tile in the start screen, pressing the details section of an alarm or reminder, and using the hub extension points. See chapter 9 for more details on tiles, and chapter 7 for hub extension points.

The first time an application instance runs, the `Launching` event is raised. When a dormant application resumes, the `Launching` event isn't raised. Because the `Launching` event is only raised once in an application's lifetime, the `Launching` event handler makes an ideal method for performing application initialization tasks.

> **NOTE** The *Application Certification Requirements for Windows Phone* require that the first page be shown in less than 5 seconds and must be responsive to user input in less than 20 seconds.

The lifetime sample application doesn't have any initialization requirements, other than to record the time when the `Launching` event is raised, which you capture in another automatic property of the `App` class. Open the App.xaml.cs file and add a new `LaunchedTime` property:

```
public DateTime LaunchedTime { get; private set; }
```

You assign the property's value in the `Launching` event handler. You learned earlier in the chapter that the event handler was automatically generated with the name `Application_Launching` by the project template. Find the generated method and assign the current time to the `LaunchedTime` property:

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    LaunchedTime = DateTime.Now;
}
```

The `Application_Launching` method is where you might put additional code that should only run once in the lifetime of an application. Be careful when performing tasks inside the `Launching` event handler. The `Launching` event is raised before the `MainPage` is created and initialized. Work performed in the event handler will increase the amount of time before the application is ready to receive user input. If the delay is too long, the operating system will terminate the application.

Now that you have the launch time recorded, you can update `MainPage` to display the value. Add another two `TextBlock` controls to MainPage.xaml:

```
<TextBlock Grid.Row="1" Text="Application launched:" />
<TextBlock x:Name="launched" Grid.Row="1" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />
```

Next, add a line to `UpdateUserInterface` to copy the value from the `App` property to the `TextBlock` control named `launched`:

```
launched.DataContext = (now - app.LaunchedTime).TotalSeconds;
```

With this code in place, run the application. You have a new line in the user interface showing launched time. At this point, showing the launch time doesn't seem very interesting. The launch time is nearly exactly the same value as the construction times. When we discuss tombstoned applications later in the chapter, you'll see situations where the launch time is greater than the construction times.

Before we can discuss tombstoning, we need to talk about what happens when the operating system deactivates, and later reactivates, an application.

3.3 Switching applications

When the user leaves an application using the Start button, the operating system pauses the application, putting the application's process into a dormant state. Other actions that cause an application to go dormant include using launchers and choosers, tapping a notification, or using the Task Switcher to change to another application.

There are situations where a dormant application is terminated, but remains on the Task Switcher's back stack. This process is named *tombstoning*. When the application enters a tombstoned state, its physical process is terminated to save system resources but its state is maintained by the operating system in volatile memory. Powering off the device or exhausting the battery has the effect to losing all the states maintained in the memory.

In this section you add features to the Lifetime sample application that detect and handle both going dormant and being tombstoned.

3.3.1 Going dormant

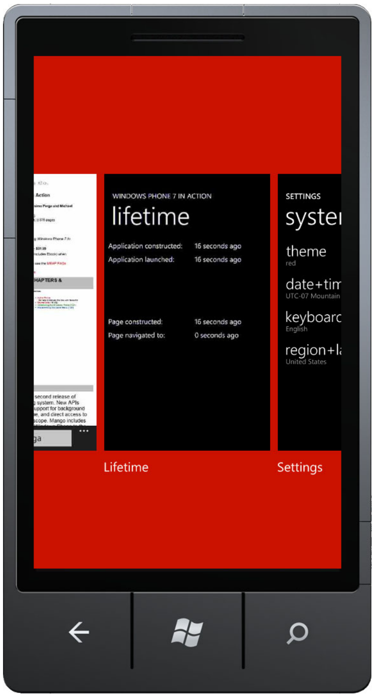
When an application is dormant, the operating system stops all running threads, unhooks sensors, and stops any timers. The application is kept in memory, is placed on the top of the stack of applications shown in the Task Switcher, and is accessible using the Back button. Figure 3.4 shows the Task Switcher with the Lifetime application in the middle of the stack.

Only five applications are kept in the back stack, including the currently running application. If there are already five applications on the back stack when a new application is started, the dormant application at the bottom of the stack is terminated. Once terminated, the application is removed from memory and any transient data used by the application is lost. The user can no longer navigate to the dormant application.

The operating system doesn't notify the dormant application when it's terminated. If you have any data that must be persisted to long term storage, you must save it before or during deactivation. We discuss local long term storage options in chapter 5.

The sample application doesn't have any data storage requirements, other than to capture the time when the `Deactivated` event is raised in a property of the `App` class. Open the App.xaml.cs file and add a new `DeactivatedTime` property:

```
public DateTime DeactivatedTime { get; private
     set; }
```



Application constructed: 16 seconds ago
Application launched: 16 seconds ago

Page constructed: 16 seconds ago
Page navigated to: 0 seconds ago

Figure 3.4 The dormant Lifetime application shown in the Task Switcher

You assign the property's value in the `Deactivated` event handler that was generated by the project template. The event handler was generated with the name `Application_Deactivated`. Find the generated method and assign the current time to the `DeactivatedTime` property:

```
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    DeactivatedTime = DateTime.Now;
}
```

The `MainPage` class also offers a hook to catch when the application is losing focus. The `OnNavigatingFrom` and `OnNavigatedFrom` override methods are called even when the operating system is navigating away from a running application. Your sample application will record when the `OnNavigatedFrom` method is called with a new `MainPage` field named `navigatedFromTime`:

```
DateTime navigatedFromTime;
```

Next you override the `OnNavigatedFrom` method and assign the new field with the current time. You only want to record the current time when navigating away from the application. You can determine whether the navigation target is external to your application by checking the `IsNavigationInitiator` property of the `NavigationEventArgs` parameter. The `IsNavigationInitiator` will be `false` if leaving the application:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (!e.IsNavigationInitiator)
    {
        navigatedFromTime = DateTime.Now;
    }
    base.OnNavigatedFrom(e);
}
```

The `IsNavigationInitiator` will be `false` anytime you're leaving the application, including when the user is closing the application using the Back key.

You need to show the new times in the user interface by adding four new `TextBlock` controls and modifying the `UpdateUserInterface` method. Open MainPage.xaml and add two `TextBlock` labels and two `TextBlock` controls named `deactivated` and `navigatedFrom`:

```
<TextBlock Grid.Row="2" Text="Application deactivated:" />
<TextBlock x:Name="deactivated" Grid.Row="2" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />

<TextBlock Grid.Row="7" Text="Page navigated from:" />
<TextBlock x:Name="navigatedFrom" Grid.Row="7" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />
```

Modify the `UpdateUserInterface` method in MainPage.xaml.cs to assign the `DataContext` properties:

```
if (navigatedFromTime != DateTime.MinValue)
    navigatedFrom.DataContext = (now - navigatedFromTime).TotalSeconds;

if (app.DeactivatedTime != DateTime.MinValue)
    deactivated.DataContext = (now - app.DeactivatedTime).TotalSeconds;
```

In this code snippet, you check whether the time values are equal to `DateTime.MinValue`. `DateTime.MinValue` is the default value for a `DateTime` variable, and you don't want to show a value on the screen if the variable still equals its default value.

Now you're ready to run the application. At first launch, you can see the new `TextBlock` controls and they don't display any values since you haven't deactivated the application. Press the Start button, wait a few seconds, and press the Back button. When the application restarts, you should now see values displayed in the `deactivated` and `navigatedFrom` controls.

We have one more event to look at: the `Activated` event, which is raised when the application restarts and returns to action.

3.3.2 Returning to action

When an application is dormant, the user can return to the application by tapping the Back button one or more times, or by selecting the application in the Task Switcher. When a dormant application is reactivated, threads are restarted and the `PhoneApplicationService` raises the `Activated` event. Because the application was never removed from memory, the `App` and `MainPage` instances are preserved and don't need to be reconstructed.

NOTE Microsoft recommends that a dormant application be ready for user input within one second after being reactivated.

It's possible that a dormant application may never return to a running state. The operating system might terminate the application if other applications are started. If the user taps an application or secondary tile in the start screen or the application's icon in the Application List, a new instance of the application will be launched and the dormant instance will be lost forever.

As with the other lifetime events, you need to record the time the event occurred in a property of the `App` class. Create a new property named `ActivatedTime` and assign the property value in the `Application_Activated` method:

```
public DateTime ActivatedTime { get; private set; }

private void Application_Activated(object sender, ActivatedEventArgs e)
{
    ActivatedTime = DateTime.Now;
}
```

Next add two more `TextBlock` controls to MainPage.xaml to display the recorded time:

```
<TextBlock Grid.Row="3" Text="Application activated:" />
<TextBlock x:Name="activated" Grid.Row="3" Grid.Column="1"
    Text="{Binding StringFormat='\{0:N0\} seconds ago'}" />
```

Finally modify the `UpdateUserInterface` method to assign the `DataContext` property of the new `TextBlock` control:

```
if (app.ActivatedTime != DateTime.MinValue)
    activated.DataContext = (now - app.ActivatedTime).TotalSeconds;
```

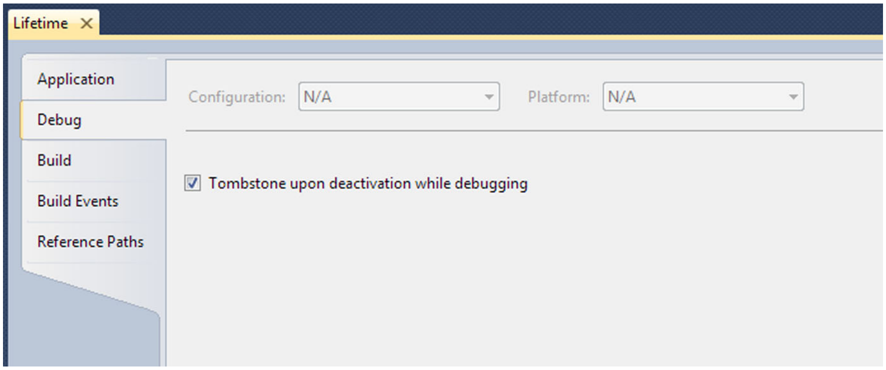
Dormant applications maintain most of their application data and state. When dormant applications are reactivated, there's usually little work to do. There are situations, especially when using the sensor APIs, when additional steps are required during activation. We'll identify these special scenarios in later chapters of the book.

There's one other reactivation process we need to discuss. In certain situations the operating system will tombstone a dormant application to free up system resources. A tombstoned application is a dormant application that has been terminated, while remaining on the Task Switcher back stack. Application developers need to write special code to save and restore application state during the tombstoning process.

3.3.3 Tombstoning

Windows Phone tombstones an application whenever system resources become scarce. To allow developers the ability to test tombstone recovery code, Microsoft has provided an option in the project's properties to force the operating system to tombstone an application. Figure 3.5 shows the tombstone option.

Enable tombstoning in the Lifetime sample project properties and debug the application. It's important to debug the application, as the tombstone option doesn't apply



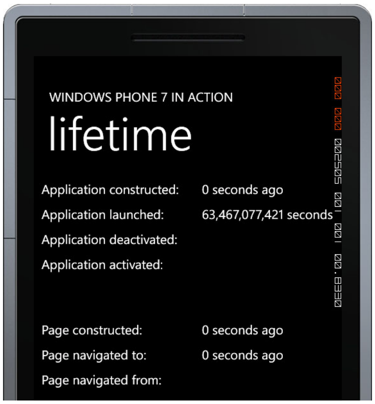
Configuration: N/A

Platform: N/A

☑ Tombstone upon deactivation while debugging

Figure 3.5 The tombstone debugging option in the project properties page

when the application is run outside the debugger. The constructed and launched times should look normal. Press the Start button, wait a few seconds, and press the Back button. The times displayed in the user interface should look weird, as shown in figure 3.6. The construction and activated times look normal, but the launch time isn't right. Note that the deactivated and navigated from times are still at zero, even though you did navigate away from this page.


WINDOWS PHONE 7 IN ACTION

lifetime

Application constructed: 0 seconds ago

Application launched: 63,467,077,421 seconds

Application deactivated:

Application activated:

Page constructed: 0 seconds ago

Page navigated to: 0 seconds ago

Page navigated from:

Figure 3.6 The Lifetime sample application after reactivated from a tombstoned state

Note that the construction times are zero seconds ago. The operating system destroyed the instances of the `App` and `MainPage` classes that were in memory. When the user pressed the Back button, new instances were created. You lost the values that had been stored in the `App` and `MainPage` properties. Some of these runtime values don't need to be saved, but others, like `DeactivatedTime`, `LaunchedTime`, and `navigatedFromTime`, are lost if you don't store them.

The Windows Phone application framework provides applications with two mechanisms for storing data that should be restored after tombstoning. The process to save and restore the state requires minimum work by the developer, which will be detailed in this section.

When a dormant application is returned to life, the `PhoneApplicationService` raises the `Activated` event. The `Activated` event is also raised when a tombstoned application is returned to life. The `ActivatedEventArgs` class provides a `boolean` `IsApplicationInstancePreserved` property to allow developers the ability to distinguish between dormant and tombstoned reactivation. You'll now show the value of this property on the user interface alongside the event times. You need add a new `nullable` `boolean` property to the `App` class to record the `IsApplicationInstancePreserved` value provided by the event args. You assign a value to the new property in the `Application_Activated` method:

```
public bool? IsApplicationInstancePreserved { get; private set; }

private void Application_Activated(object sender, ActivatedEventArgs e)
{
    ActivatedTime = DateTime.Now;
    IsApplicationInstancePreserved = e.IsApplicationInstancePreserved;
}
```

Next you modify MainPage.xaml to add controls to display the value:

```
<TextBlock Grid.Row="4" Text="App Instance Preserved:" />
<TextBlock x:Name="instancePreserved" Grid.Row="4" Grid.Column="1" />
```

Finally you modify the `UpdateUserInterface` method in MainPage.xaml.cs to assign the `Text` property of the `instancePreserved` `TextBlock`:

```
instancePreserved.Text = app.IsApplicationInstancePreserved.ToString();
```

Debug the application now and you should see a blank in the new control because the `nullable` boolean property defaults to `null`. Press the Start button to navigate away and then return to the application using the Back button. You should now see the word False in the `instancePreserved` `TextBlock`. If you run the application outside the debugger and perform the same steps, the `TextBlock` should display the word True.

Now that you know how to detect when the application has been tombstoned, how do you save the `DeactivatedTime`, `LaunchedTime`, and `navigatedFromTime` values? The first facility for storing application data is the `State` property exposed on the `PhoneApplicationService`. The `State` property is a dictionary of key-value pairs and will hold any object that can be serialized. Information should be placed into the `State` dictionary when the application is being deactivated. Modify the `Application_Deactivated` method to store the `DeactivatedTime` and `LaunchedTime` in the `State` dictionary:

```
private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    DeactivatedTime = DateTime.Now;
    PhoneApplicationService.Current.State["DeactivatedTime"]
        = DeactivatedTime;
    PhoneApplicationService.Current.State["LaunchingTime"] = LaunchedTime;
}
```

Application data stored in the `State` dictionary can be restored in the `Activation` event handler. Update the `Application_Activated` method to restore the `DeactivatedTime` and `LaunchedTime` values. The following listing shows the new `Application_Activated` method.

Listing 3.4 Restoring tombstoned application data

```
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    ActivatedTime = DateTime.Now;
    IsApplicationInstancePreserved = e.IsApplicationInstancePreserved;
    if (!e.IsApplicationInstancePreserved)
    {
        if (PhoneApplicationService.Current.State.
            ContainsKey("DeactivatedTime"))
        {
            DeactivatedTime = (DateTime)PhoneApplicationService.Current.
                State["DeactivatedTime"];
        }
        if (PhoneApplicationService.Current.State.
            ContainsKey("LaunchingTime"))
        {
            LaunchedTime = (DateTime)PhoneApplicationService.Current.
                State["LaunchingTime"];
        }
    }
}
```

❶ **Was app tombstoned?**

❷ **Restore DeactivatedTime**

❸ **Restore LaunchedTime**

The `DeactivatedTime` and `LaunchedTime` values should only be restored when recovering from a tombstoned state. You only read from the `State` dictionaries when `IsApplicationInstancePreserved` is false ❶. Next you ask the `State` dictionary whether it contains a value for the key `DeactivatedTime`. If the key exists you assign the value in the dictionary to the `DeactivatedTime` property ❷. You do the same check and assign operation for `LaunchedTime` ❸.

The second facility for storing runtime application data is another `State` dictionary provided by the `PhoneApplicationPage`. Because `PhoneApplicationPage` is the base class for `MainPage`, you can store `MainPage`'s runtime data separate from the `App` class's runtime data. The only `MainPage` data you need to store is the `navigatedFromTime`. Update the `OnNavigatedFrom` method to store the `navigatedFromTime` right after assigning the variable:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (!e.IsNavigationInitiator))
    {
        navigatedFromTime = DateTime.Now;
        State["NavigatedFromTime"] = navigatedFromTime;
    }
    base.OnNavigatedFrom(e);
}
```

The sample application writes data to the `State` dictionary during the `OnNavigatedFrom` method, but you don't have to wait until `OnNavigatedFrom` is called. The `State` dictionary can be accessed anytime during or after `OnNavigatedTo` and before or during `OnNavigatedFrom`. Objects stored in either of the `State` dictionaries must be serializable.

The `navigatedFromTime` is restored in the `OnNavigatedTo` method. The new and improved `OnNavigatedTo` method is shown in the next listing.

Listing 3.5 Restoring `navigatedFromTime`

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    navigatedToTime = DateTime.Now;

    var app = (App)Application.Current;
    bool appInstancePreserved =
        app.IsApplicationInstancePreserved ?? true;

    if (!appInstancePreserved && State.ContainsKey("NavigatedFromTime"))
    {
        navigatedFromTime = (DateTime)State["NavigatedFromTime"];
    }

    UpdateUserInterface();
    base.OnNavigatedTo(e);
}
```

① **Check for tombstone**

Restore navigatedFromTime ②

The only way to determine whether you're resuming from a tombstoned state is via the `IsApplicationInstancePreserved` property of the `ActivatedEventArgs` passed with the `Activated` event. You stored the `IsApplicationInstancePreserved` value in a property of the `App` class, which you use to determine whether the application is resuming from a tombstoned state ①. If it was tombstoned, and the `State` dictionary contains the `NavigatedFromTime` key, then you assign the value stored in the dictionary to the `navigatedFromTime` field ②.

> **NOTE** If you're unfamiliar with C#'s `??` operator, which is called the *null-coalescing operator*, it's used to assign a default value to a variable. In listing 3.5, if the `app.IsApplicationInstancePreserved` property is `null`, the `appInstancePreserved` variable will be assigned the value after the ?? operator, which is `true` in this example.

Debug the application again, press the Start button, wait a few seconds, and press the Back button. The times displayed in the user interface should look almost normal this time. Note that the launching and navigated from times are further in the past than the construction times. This is expected because the `App` and `MainPage` instances were recreated when the application was activated, but the launching and navigated from times were saved in the `State` dictionaries.

We've demonstrated a couple of procedures for saving a restoring application and page state when the operating system reactivates your application. It's important that the save and restore routines execute reasonably quickly, as the operating system will kill any application that takes longer than 10 seconds to respond to activated or deactivated events.

The `Launching`, `Activated`, and `Deactivated` events are all notifications that the application is transitioning into or out of a running state. The last two events we look at, `Obscured` and `Unobscured`, are notifications that the application screen is partially or fully hidden, but that the application remains in the running state.

3.4 Out of sight

As you design applications for a phone, you must prepare for situations when the operating system will notify the user about an incoming call or a message. This situation is named *obscuration* and in practice this is a partial or full coverage of an application made by a native application. Your application will also be obscured when an alarm or reminder is triggered, and in special circumstances, when the phone's lock screen is activated. In this section we first examine how to detect when an application is obscured, and then we discuss the special circumstances surrounding the lock screen.

3.4.1 Obscuration

The event raised when the application gets covered is named `Obscured` and the event raised when the application is again fully visible is named `Unobscured`. Both these events belong to the `PhoneApplicationFrame` class, and the developer will have to explicitly add handlers for these events because they're not automatically generated by the Visual Studio wizard.

Many applications aren't affected by obstruction and can ignore the obscuration events. Other applications are more sensitive. An application that plays video might want to pause playback and then resume when the screen is no longer obscured. A Silverlight game should pause game play and/or game timers so a user isn't penalized when the screen is inaccessible.

The `App` class defines the only `PhoneApplicationFrame` instance as the member variable named `RootFrame`. The next listing shows how to wire up and implement the event handlers in the `MainPage` class.

Listing 3.6 Adding `Obscured` and `Unobscured` event handlers

```
DateTime obscuredTime;                                          ① Define
DateTime unobscuredTime;                                          new fields

public MainPage()
{
    InitializeComponent();
    pageConstructedTime = DateTime.Now;

    ((App)Application.Current).RootFrame.Obscured +=
        RootFrame_Obscured;                                      ② Subscribe
    ((App)Application.Current).RootFrame.Unobscured +=             to events
        RootFrame_Unobscured;
}

void RootFrame_Unobscured(object sender, EventArgs e)
{
    unobscuredTime = DateTime.Now;                             ③ Show
    UpdateUserInterface();                         <                updated time
}

void RootFrame_Obscured(object sender, ObscuredEventArgs e)
{
    obscuredTime = DateTime.Now;
    UpdateUserInterface();
}
```

Two new fields are added to the `MainPage` class ❶ to store the time when the `Obscured` and `Unobscured` events are raised. The events are wired up in the `MainPage` constructor ❷. The `App` class's `RootFrame` property is used to access the `PhoneApplicationFrame` instance. In each of the event handlers, the time is assigned to the appropriate field and the user interface is updated to show the new times ❸.

We leave it as an exercise for the reader to modify `UpdateUserInterface` so that `obscuredTime` and `unobscuredTime` values are displayed in new user interface controls.

In this section we've demonstrated how an application becomes dormant when the user presses the Start button. Another scenario that pauses a running application is when the phone is locked and the lock screen is activated. An application isn't required to go dormant when the phone is locked and may choose to remain running behind the lock screen.

3.4.2 Running behind the lock screen

All Windows Phones have a lock button the user can use to lock the phone and power off the device. A phone will also automatically lock after a specified time-out period in which the phone remains idle. The screen time-out duration is specified in the lock+wallpaper page of the Settings application. The phone is considered idle when the user hasn't tapped the touch screen.

> **NOTE** The emulator doesn't provide a lock button and doesn't activate the lock screen. If your application runs behind the lock screen, you must test your code on a physical device.

When the phone is locked, the running application transitions to a dormant state. To see this in action, run the application and lock the screen. When you unlock the screen, you'll find that the application was deactivated while the screen was locked, then reactivated when the screen was unlocked. Note that the `ActivatedTime` is zero seconds ago. This is a feature of the operating system intended to save on battery life.

The `PhoneApplicationService` provides the `ApplicationIdleDetectionMode` property to allow applications to continue to run once the phone is locked. When an application wants to run under the lock screen, the `ApplicationIdleDetectionMode` property is assigned the value `IdleDetectionMode.Disabled`. Once idle detection mode is disabled, it can't be re-enabled. An attempt to re-enable idle detection will result in an exception.

> **NOTE** The *Application Certification Requirements for Windows Phone* place certain restrictions on applications that run behind the lock screen. Once the screen is locked, the application should no longer attempt to update the user interface and should disable all timers. The battery must be able to power the phone for 120 hours while the application is running under the lock screen.

An application is notified when the phone is locked via the `Obscured` event. The `Obscured` event handler passes an argument of type `ObscuredEventArgs`. The `ObscuredEventArgs` contains a `boolean` property named `IsLocked`. When `IsLocked` is true, the application is running behind the lock screen.

A good application will inform the user that it runs behind the lock screen. A better application will allow the user to choose whether the application runs behind the lock screen. Modify the Lifetime sample application to provide the user the ability to enable running behind the lock screen. Start by adding a check box to MainPage.xaml:

```
<CheckBox x:Name="runOption" Margin="0,24,0,0"
    Grid.Row="10" Grid.ColumnSpan="2"
    Content="Run while the screen is locked."
    Checked="runOption_Checked" IsChecked="False" />
```

The `CheckBox` `IsChecked` property is initially set to `False`. The `Checked` event is wired up to an event handler named `runOption_Checked`. Inside the event handler, set the `ApplicationIdleDetectionMode` to `Disabled`:

```
private void runOption_Checked(object sender, RoutedEventArgs e)
{
    PhoneApplicationService.Current.ApplicationIdleDetectionMode
        = IdleDetectionMode.Disabled;
    runOption.IsEnabled = false;
}
```

Because the `ApplicationIdleDetectionMode` property cannot be re-enabled, you disable the `CheckBox` control.

To see how the new code changes the behavior of the Lifetime sample application, run the application and tap the new `CheckBox`. Next lock the phone, wait a few seconds, and unlock the phone. The user interface should show blanks for the deactivated and activated event times, and show values for obscured and unobscured times.

This wraps up the first sample application you built in this chapter. The Lifetime sample demonstrated how an application lives and dies. During an application's lifetime, it may become dormant or be tombstoned when the user switches to another application. We showed you how to use the lifetime events to detect transitions between living, pausing, and dying. You learned how to use the `State` dictionaries to save and restore application state in the event when the application is tombstoned. Understanding the lifetime events is key to building an application that works well with fast application switching. Fast application switching gives the user the impression that an application remains running in memory.

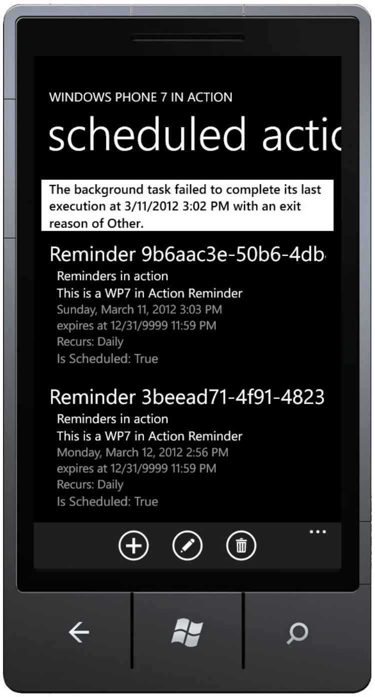
You now know that a dormant background application can't perform any work. How can you create killer applications if they can't do any work in the background? The answer lies with the Scheduled Action Service. In the next section you'll create a new sample application to explore alarms, reminders, and background tasks, three different kinds of scheduled actions. Scheduled actions allow your application to alert the user or execute background work when your application isn't running.

3.5 Working on a schedule

A great number of use cases require an application to perform work on a periodic basis. This work might entail reminding the user that an online auction is about to close so they can log on and ensure they're the top bidder. Another example might be a CRM application that checks in with a web service to download new sales leads to the device.

Windows Phone empowers developers to build these types of applications with scheduled actions and the *Scheduled Action Service*. Scheduled actions are named actions that have beginning and expiration times. Once a scheduled action is registered, the Scheduled Action Service will execute the action at the appropriate time.

Two forms of scheduled actions are provided—notifications and tasks. Notifications consist of alarms and reminders and are displayed to the user at the appropriate time. The user can dismiss or snooze notifications, and can tap the content of the notification to launch the application that created them. A scheduled task is nothing more than a request that the Scheduled Action Service launch an application's background agent, allowing the application to perform background processing.


WINDOWS PHONE 7 IN ACTION

scheduled actio

The background task failed to complete its last execution at 3/11/2012 3:02 PM with an exit reason of Other.

Reminder 9b6aac3e-50b6-4db
Reminders in action
This is a WP7 in Action Reminder
Sunday, March 11, 2012 3:03 PM
expires at 12/31/9999 11:59 PM
Recurs: Daily
Is Scheduled: True

Reminder 3beead71-4f91-4823
Reminders in action
This is a WP7 in Action Reminder
Monday, March 12, 2012 2:56 PM
expires at 12/31/9999 11:59 PM
Recurs: Daily
Is Scheduled: True

Figure 3.7 The ScheduledActions sample application

In this section you'll create a new sample application that you'll use throughout the remainder of this chapter. A screenshot of the completed sample application is shown in figure 3.7. The sample application will create, update, and delete scheduled actions. Later in the chapter you'll add a background agent to the application that will monitor and delete expired alarms and reminders.

Start the new sample application using the Windows Phone Application project template, name the project ScheduledActions, and be sure to select version 7.1 as the target operating system. Open up MainPage.xaml in the editor so that you can add the basic user interface elements displayed by the application. Start by dividing the `ContentPanel` `Grid` control into two rows:

```
<Grid x:Name="ContentPanel" Grid.Row="1"
    Margin="12,0,12,0">
   <Grid.RowDefinitions>
       <RowDefinition Height="Auto" />
       <RowDefinition Height="*" />
   </Grid.RowDefinitions>
</Grid>
```

You'll use the first row to display messages about the background agent later in the chapter. Add a `ListBox` to the second row of the `ContentPanel` (you'll add controls to display details of each notification to the `DataTemplate` later):

```
<ListBox x:Name="notificationList" Grid.Row="1">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Margin="12">
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

The application displays three buttons in the `ApplicationBar`. Listing 3.7 shows the `ApplicationBar` markup for MainPage.xaml. Several images are used for the application bar and must be added to the project. Create a project folder named Images and copy the image files named appbar.add.rest.png, appbar.delete.rest.png, and appbar.edit.rest.png from the Windows Phone 7.1 SDK. The images files can be found in C:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Icons\dark or C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Icons\dark. Remember to set each image file's build action property to `Content`.

Listing 3.7 `ApplicationBar` markup for the ScheduledActions sample application

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton
            IconUri="/Images/appbar.add.rest.png"
            Text="reminder" Click="AddReminder_Click" />
        <shell:ApplicationBarIconButton
            IconUri="/Images/appbar.edit.rest.png"
            Text="reschedule" Click="RescheduleNotification_Click" />
        <shell:ApplicationBarIconButton
            IconUri="/Images/appbar.delete.rest.png"
            Text="remove" Click="RemoveNotification_Click" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

① **Button image**

② **Click event handler**

This code listing adds three buttons to the application bar. Each button uses one of the images ① you just added to the project. Each button wires up its `Click` event ② to an event handler in the code behind. For the time being, add an empty event handler to MainPage.xaml.cs. You'll implement the event handlers as you work through the sample.

The sample application's skeleton is now in place and ready for you to create, update, and delete scheduled actions. Each of these operations is performed using the Scheduled Action Service.

3.5.1 Introducing the Scheduled Action Service

The Scheduled Action Service is implemented as a singleton by the `ScheduledActionService` class and is found in the `Microsoft.Phone.Scheduler` namespace. The class provides methods for adding, updating, and removing scheduled actions. Scheduled actions are retrieved from the service either as a list or individually by name. Scheduled actions can only be seen and modified by the application that created them. The scheduled action service won't expose actions created by other applications.

Scheduled actions are defined by the `ScheduledAction` class. The `ScheduledAction` class is the base class for both the `ScheduledNofitication` class and the `ScheduledTask` class. These last two classes are base classes for other scheduled action classes we look at later in the chapter. Table 3.2 describes each of the properties exposed by the `ScheduledAction` class.

Table 3.2 `ScheduledAction` properties

Property name	Description
`BeginTime`	The `DateTime` when the action will be triggered for the first time. `BeginTime` must represent some point in the future when the action is scheduled.
`ExpirationTime`	The `DateTime` after which the action will no longer be triggered. `ScheduledNotifications` that have been snoozed by the user won't be triggered after the expiration time has passed. The expiration time defaults to `DateTime.MaxValue`. `ExpirationTime` must be greater than `BeginTime` when the action is scheduled.
`IsEnabled`	A read-only property, `IsEnabled` will always be true for `ScheduledNotifications`. `IsEnabled` will be false for `ScheduledTasks` if the user has disabled background tasks for the application.
`IsScheduled`	A read-only property, `IsScheduled` will be true if the `ScheduledAction` will be invoked at some point in the future. `IsScheduled` with be false if the user disables a task or dismisses a non-recurring notification, or when the action's `ExpirationTime` has passed. Recurring notifications will remain scheduled when the user dismisses the notification.
`Name`	A unique identifier for the `ScheduledAction`.

The sample application displays every scheduled notification registered with the Scheduled Action Service. The application invokes the `GetActions` method which returns a collection of actions. The collection of actions is then displayed in the `ListBox` you added to MainPage.xaml. The code to retrieve the actions is shown in the following listing.

Listing 3.8 Building a list of notifications

```
protected void DisplayScheduledNotifications()
{
    var items = new List<ScheduledAction>();
    var notifications = ScheduledActionService.
        GetActions<ScheduledNotification>();
    foreach (var notification in notifications)
    {
        var item = ScheduledActionService.Find(notification.Name);
        items.Add(item);
    }
    notificationList.ItemsSource = items;
}
```

① **Get alarms and reminders**

② **Use instance returned by Find**

First you define a new `List` to contain the alarms and reminders that will be displayed in the user interface. You obtain a collection of notifications from the Scheduled Action Service by calling the `GetActions` method. `GetActions` is a generic method and expects the calling code to declare which type of scheduled action is to be returned. The sample application displays both alarms and reminders, so the code invokes `GetActions` with the `ScheduledNotification` ❶ type, the base class for both the `Alarm` and `Reminder` classes. Next you obtain another reference to the notification by calling the `Find` method ❷ with the name of the target item. You add each item returned by `Find` to the items list. Finally you assign the items list to the `ListBox`'s `ItemSource` property.

> **TIP** The Scheduled Action Service has an odd behavior—the notifications returned by `GetActions` are cached clones. Calling `GetActions` returns the cached instances. If an action's state has changed, such as when an alarm is shown to and dismissed by the user, the cached copy isn't updated. When the `Find` method is called, the cache is updated and the most recent state of the action is available. You're calling `Find` within the `DisplayScheduledNotifications` method to force an update of the cache.

The new `DisplayScheduledNotifications` method will be invoked in a few different spots within your application. The method should be invoked every time `MainPage` is navigated to, so you add a call `DisplayScheduledNotifications` in a new override of the `OnNavigatedTo` method:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    DisplayScheduledNotifications();
    base.OnNavigatedTo(e);
}
```

In the last section, you added a `DataTemplate` to the `ListBox`, but only declared an empty `StackPanel` inside the template. You need to add user interface components to display the properties of the scheduled notifications that were added to the `ListBox`. The following listing contains the fully declared `DataTemplate`.

Listing 3.9 Displaying a notification

```
<DataTemplate>
    <StackPanel Margin="12">
        <TextBlock Text="{Binding Name}"
                   FontSize="{StaticResource PhoneFontSizeLarge}" />
        <TextBlock Text="{Binding Title}"
                   Style="{StaticResource PhoneTextNormalStyle}" />
        <TextBlock Text="{Binding Content}"
                   Style="{StaticResource PhoneTextNormalStyle}" />
        <TextBlock Text="{Binding BeginTime, StringFormat='\{0:f\}'}"
                   Style="{StaticResource PhoneTextSmallStyle}" />
        <TextBlock Text="{Binding ExpirationTime,
➥ StringFormat='expires at \{0\:g}'}"
```

❶ **Add TextBlocks to StackPanel**

Format fields with StringFormat ❷

```
                   Style="{StaticResource PhoneTextSmallStyle}" />
        <TextBlock Text="{Binding RecurrenceType,
➥  StringFormat='Recurs: \{0\}'}"
                   Style="{StaticResource PhoneTextSmallStyle}" />
        <TextBlock Text="{Binding IsScheduled,
➥  StringFormat='Is Scheduled: \{0\}'}"
                   Style="{StaticResource PhoneTextSubtleStyle}" />
    </StackPanel>
</DataTemplate>
```

The `DataTemplate` contains several `TextBlock` instances ❶, each one displaying a different `ScheduledAction` property. The `TextBlocks` are styled using theme resources provided by the Silverlight framework. Some of the `TextBlocks` make use of the `StringFormat` binding markup ❷ to display in a more user-friendly format.

You now have a user interface that will display scheduled notifications to the user. At this point, the user interface remains blank, because there are no scheduled notifications to display. You need to implement the code to create some alarms and reminders.

3.5.2 Scheduling a reminder

Scheduled notifications are alarms and reminders that are registered with the operating system. Due to limitations imposed by the Windows Phone operating system, an application can't directly notify the user unless the application is running. With a scheduled notification, an application can schedule an alarm or a reminder and know that the user will be notified at the appropriate time, even if the application isn't running.

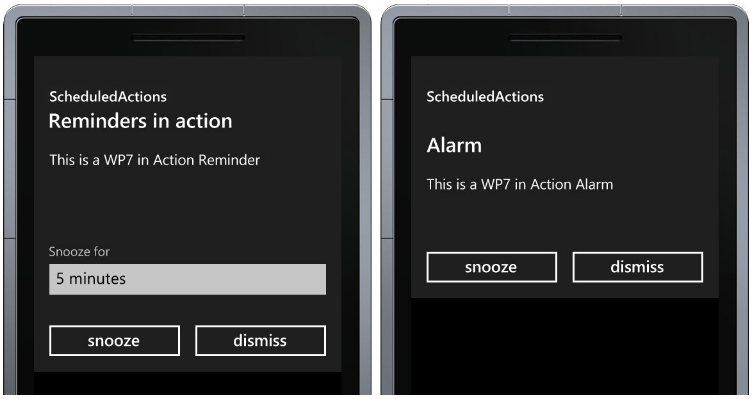
Scheduled notifications are implemented in two related classes named `Alarm` and `Reminder`. Both the `Alarm` and `Reminder` classes are derived from the `ScheduledNotification` class. As mentioned earlier, `ScheduledNotification` derives from `ScheduledAction`. `ScheduledNotification` extends its base class with additional properties named `Content` and `RecurrenceType`.

The `Content` property allows the application a place to display a message to the user when the notification is triggered. The `Content` property is a string with a maximum length of 256 characters.

The `RecurrenceType` property allows an application to create notifications that are triggered more than once. The recurrence patterns defined in the Scheduler API include daily, weekly, monthly, and yearly patterns. A notification is triggered at the appropriate time interval after the `BeginTime`. For example, if the recurrence pattern is daily and the `BeginTime` is specified as 8:00 a.m. Monday, July 4th, the notification will be triggered every day at 8:00 a.m. until the `ExpirationTime` has passed.

> **NOTE** The ScheduledActions sample application you build here will only create reminders. Creating alarms follows a similar pattern and we leave it as an exercise for the reader to add alarms to the project.

The ScheduledActions sample application defines a button allowing the user to create a `Reminder`. The click event handler for the Add Reminder button creates a new instance of a `Reminder` and registers it with the scheduled action service. The right side of figure 3.8


ScheduledActions
Reminders in action
This is a WP7 in Action Reminder
Snooze for
5 minutes

ScheduledActions
Alarm
This is a WP7 in Action Alarm

Figure 3.8 A reminder and an alarm as displayed to the user

shows what the user will see when a reminder is triggered. For comparison, we show an alarm on the left side of the figure. When the user taps the title or content of a notification, the operating system will launch the host application's main page.

The observant reader will notice that a `Reminder` displays a custom title, whereas an Alarm always displays *Alarm* as the title. The title displayed by the `Reminder` is specified using the `Title` property. Another difference between `Alarms` and `Reminders` is that `Reminders` can specify a `NavigationUri` property, whereas `Alarms` have a `Sound` property.

The `Alarm.Sound` property is an `Uri` to any supported audio file located in the application's .xap file. The sound file must added to the project with its build action property set to `Content`.

The `Reminder.NavigationUri` property is used by the operating system when the user taps the title or content of a `Reminder`. The operating system will launch the host application, but will use the `NavigationUri` as the starting page instead of the default page. Listing 3.10 demonstrates how a `Reminder` is constructed and registered in the Add Reminder button's click event handler.

Listing 3.10 Creating a reminder

```
private void AddReminder_Click(object sender, EventArgs e)
{
    string reminderName = string.Format("Reminder {0}", Guid.NewGuid());

    Reminder reminder = new Reminder(reminderName);
    reminder.BeginTime = DateTime.Now.AddMinutes(1);
    reminder.RecurrenceType = RecurrenceInterval.Daily;
    reminder.Content = "This is a WP7 in Action Reminder";
    reminder.Title = "Reminders in action";
    reminder.NavigationUri = new Uri(
        "/MainPage.xaml?reminder=" + reminderName, UriKind.Relative);

    ScheduledActionService.Add(reminder);
    DisplayScheduledNotifications();
}
```

❶ Repeat every day

❷ Set title

❸ Add navigation path

`Reminders` must have unique names, and you start by creating a name containing a `Guid`. Next you construct a new `Reminder` and assign several of its properties. You hard-code the `BeginTime` to be one minute from the current time. Remember that the begin time must always be in the future. You also specify that the alarm should repeat at the same time every day ❶ by assigning a `RecurrenceType` of `Daily`. You assign the `Title` property ❷, and construct a `NavigationUri` ❸ so that the `Reminder` navigates to the `MainPage` and includes the reminder name. Finally you add the `Reminder` to `ScheduledActionService` and update the user interface.

> **TIP** A real application might allow the user to select the `BeginTime` using date and time pickers. Date and time pickers are demonstrated in chapter 11.

When the user clicks the reminder title, the application will be launched with the specified navigation URI. Our sample doesn't define a special page and doesn't perform any special processing when receiving an URI. Your applications may choose to display the reminder details in a special page, or trigger other customized application logic when it receives a `Reminder`'s navigation URI.

You now know how to create scheduled notifications. Creation is just one of the create, update, and delete operations provided by the `ScheduledActionService`. The next operation we examine is updating or editing an existing notification.

3.5.3 Editing a notification

Once created, `ScheduledNotifications` can be modified by an application. Notifications are modified using the `Replace` method provided by the `ScheduledActionService`. The `Replace` method accepts an instance of `ScheduledNotification` and the notification's `Name` property identifies which `ScheduledNotification` is to be replaced. The `ScheduledActionService` overwrites the saved notification with the values specified in the passed-in notification.

The sample application provides a Reschedule Notification button. When the user taps the button, the notification selected in the `ListBox` will be updated with a new `BeginTime` and a modified `Content` property. The next listing details the Reschedule Notification button's click event handler.

Listing 3.11 Rescheduling a notification

```
private void RescheduleNotification_Click(object sender, EventArgs e)
{
    var notification =
        notificationList.SelectedItem as ScheduledNotification;
    if (notification != null)
    {
        notification.BeginTime = DateTime.Now.AddMinutes(1);
        notification.Content = "*" + notification.Content;

        ScheduledActionService.Replace(notification);
        DisplayScheduledNotifications();
    }
}
```

❶ Do no work if nothing is selected

❷ Update selected notification

Before doing any work, the event handler asks the `ListBox` for the currently selected item. If the `ListBox` doesn't have a selected item ❶, the event handler doesn't perform any work. When a scheduled notification is selected, its `BeginTime` property is set to be one minute in the future. Even when using the `Replace` method, the `BeginTime` must be in the future, and `ExpirationTime` must be greater than `BeginTime`. If `BeginTime` is in the past, or `ExpirationTime` is less than `BeginTime`, an exception will be thrown. After the notification is updated, the `Replace` method is executed ❷ and the user interface is updated.

When you created the sample application, you created four `ApplicationBar` buttons. You've only implemented three of the four buttons. The final feature allows the user to delete a notification.

3.5.4 Deleting a notification

There's one more button that doesn't have an event handler defined for it: the Delete Notification button. Notifications are deleted with the `Remove` method of the `ScheduledActionService`. The `Remove` method accepts the name of the notification to remove. The sample application removes a notification in the Remove Notification button's click event handler, shown in the next listing.

Listing 3.12 Deleting a notification

```
private void RemoveNotification_Click(object sender, EventArgs e)
{
    var notification = notificationList.SelectedItem
        as ScheduledNotification;
    if (notification != null)
    {
        ScheduledActionService.Remove(notification.Name);
        DisplayScheduledNotifications();
    }
}
```

❶ **Removing notification**

The application removes the notification that's selected in the `ListBox`. If the `ListBox` doesn't have a notification selected, the method does no work. If a notification is selected, the `Remove` method is called with the notification's name ❶. Finally, `DisplayScheduledNotifications` is called to refresh the user interface.

Once created, a `ScheduledNotification` will be managed by the `ScheduledActionService` until the application removes the notification or the application is uninstalled. `ScheduledNotifications` are also removed if Visual Studio's Rebuild Solution option is used to build the project. You could easily add a method to the application launching or activation routines that looked for expired and unscheduled notifications and automatically removed them. Instead we'll employ this use case to demonstrate how to create a software agent to perform work in the background, even when the host application is not running.

3.6 Creating a background agent

Sometimes an application needs to run in the background to be useful. The Windows Phone allows an application to execute certain kinds of tasks in the background. Background tasks include playing or streaming audio, as well as customs tasks that execute periodically. Background processes are named *background agents*. In this section we show you how to create a scheduled task agent to perform work periodically in the background, even when the application isn't running. We look at the audio background agents in chapter 7.

An application's scheduled task agent can run two types of scheduled tasks, both of them represented by classes derived from `ScheduledAction`. The first type of scheduled task executes once every half hour and is defined by the `PeriodicTask` class. The second type of background process is defined by the class named `ResourceIntensiveTask`. Resource-intensive tasks only execute if the device is plugged in and fully charged, connected to Wi-Fi or a computer, and screen locked. Both `PeriodicTask` and `ResourceIntensiveTask` are derived from `ScheduledTask`, which is derived from `ScheduledAction`.

Scheduled tasks aren't necessarily run exactly 30 minutes apart. The Scheduled Action Service coordinates the timing and execution of scheduled tasks. Waking up the device and powering on the sensors and radios is an expensive process. The scheduled action service will power up once, then execute all scheduled tasks, and power down. This is one of the methods employed by the Windows Phone to maximize battery life.

An application schedules a `ScheduledTask` using the `ScheduledActionService` using a process similar to scheduling a `ScheduledNotification`. An application can schedule only one `PeriodicTask` and one `ResourceIntensiveTask`. Unlike a `ScheduledNotification`, the user isn't notified when a `ScheduledTask` is triggered. Instead the operating system notifies the host application through a `ScheduledTaskAgent`. `ScheduledTaskAgents` are defined in background agent projects.

Scheduled task agents are limited in the types of work they can do. They aren't allowed to access the camera, radio, accelerometer, compass, or gyroscope. Scheduled task agents can't add new scheduled actions or new background file transfers, or show launchers or choosers. A complete list of unsupported APIs and restrictions can be found in Microsoft's SDK documentation on MSDN.

You're going to create a `ScheduledTaskAgent` that executes a `PeriodicTask`. Periodic tasks and resource-intensive tasks differ only in the schedule they run on and the amount of time they're allowed to execute.

3.6.1 Background agent projects

The ScheduledActions sample application needs a background agent to execute a periodic task that scans all scheduled notifications and removes any that are expired or aren't scheduled. To add a background agent, you need to add a new project to the ScheduledActions solution. Use the new project wizard and select the Windows Phone

Scheduled Task Agent project template. Name the new project NotificationsCleanupAgent and be sure to select the Add to Solution option in the wizard.

The new project will contain a single source code file named ScheduledAgent.cs. Open up the ScheduledAgent.cs and find the class named `ScheduledAgent`. The generated `ScheduledAgent` class is derived from the `ScheduledTaskAgent`. `ScheduledTaskAgent` is one of three classes derived from the base `BackgroundAgent` class. The other background agent classes are used for playing audio files and are covered in more depth in chapter 7.

The next step is to reference the new background agent project in the ScheduledActions project. Using the Add Reference dialog, add a project reference to the NotificationsCleanupAgent project from the ScheduledActions project. In addition to adding the project reference, Visual Studio updated the WMAppManifest.xml file so that it contains a new `ExtendedTask` element:

```
<ExtendedTask Name="BackgroundTask">
    <BackgroundServiceAgent Specifier="ScheduledTaskAgent"
        Name="NotificationsCleanupAgent"
        Source="NotificationsCleanupAgent"
        Type="NotificationsCleanupAgent.ScheduledAgent" />
</ExtendedTask>
```

Along with generated diagnostic code, the newly added `ScheduledAgent` contains a method named `OnInvoke`. `OnInvoke` is called by the Windows Phone framework to inform the agent that a `ScheduledTask` has been triggered. The agent will be passed the triggered `ScheduledTask` object. The same background agent can execute both periodic tasks and resource-intensive tasks.

3.6.2 Executing work from the background agent

You're building a background agent to remove expired notifications. The expired notifications will be removed by the `ScheduledAgent` class generated by the project template. Open the ScheduledAgent.cs file, find the `OnInvoke` method, and add code to look for and remove any expired notifications. The method implementation is shown in the following listing.

Listing 3.13 Removing expired notifications with the background agent

```
protected override void OnInvoke(ScheduledTask task)
{
    DateTime now = DateTime.Now;
    var notifications = ScheduledActionService.
        GetActions<ScheduledNotification>();
    foreach (var notification in notifications)
    {
        if (notification.IsScheduled == false
            || notification.ExpirationTime < now)
        {
            ScheduledActionService.Remove(notification.Name);
        }
```

① **Is notification expired?**

```
    }
    NotifyComplete();                                  ❷ Notify OS about
}                                                         successful completion
```

The list of all `ScheduledNotifications` is retrieved using the `GetActions` method. You iterate through the returned list looking at each notification's `IsScheduled` property and `ExpirationTime` value ❶. If a notification isn't scheduled or is expired, you use the `Remove` method to delete it from the scheduled action service. Finally, you tell the operating system that you're done by calling `NotifyComplete` ❷.

With the scheduled task agent implemented and ready to do work, you need something that will trigger the background agent. Scheduled task agents are triggered by `ScheduledTasks`. In the next section you learn how to create a `PeriodicTask` from within the ScheduledActions application.

3.6.3 Scheduling a PeriodicTask

As with `ScheduledNotifications`, a `PeriodicTask` is scheduled with the scheduled action service. Periodic tasks are instantiated, have their properties set, and are scheduled by invoking the `Add` method. `ScheduledTask` extends `ScheduledAction` by adding a `Description` property. The `Description` property is shown to the user in the background tasks setting application. The `ScheduledTask` doesn't define the code that's executed when the task is triggered—that code is defined by the background agent.

The sample application schedules the periodic task when the application is launched. You learned earlier in the chapter that the `PhoneApplicationService` raises a `Launching` event when a new instance of an application is launched by the operating system. The `App` class generated by the project template contains a `Launching` event handler named `Application_Launching`. Open App.xaml.cs and modify the `Application_Launching` event hander to create a new `PeriodicTask`, as detailed in the following listing.

Listing 3.14 Creating a `PeriodicTask`

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    PeriodicTask cleanupTask;

    try
    {
        cleanupTask = new PeriodicTask("NotificationCleanupTask");
        cleanupTask.Description = "A background agent responsible for
➥ removing expired Windows Phone 7 in Action notifications";
        ScheduledActionService.Add(cleanupTask);             Assign
    }                                                        description ❶
    catch (InvalidOperationException)
    {
        AgentStatus += "Unable to create the background task.";   Report
        cleanupTask = null;                                     ❷ failure
    }
}
```

Creating a `PeriodicTask` should feel familiar to you by now, since `PeriodicTasks`, `Alarms`, and `Reminders` are all a form of `ScheduledAction`. A `Guid` isn't included in the task's name, since the application will only create a single `PeriodicTask`. `ScheduledTask` extends `ScheduledAction` with a `Description` property, which is shown to the user in the phone's settings application. You assign the `Description` property with text that describes the work performed by the background task ❶. The call to the `Add` method is surrounded by a `try/catch` block to catch any exceptions that might be thrown, particularly when the user has permanently disabled background tasks for the applications. If an exception is thrown, you report the failure ❷ by assigning the `AgentStatus` property.

NOTE We're using the += operator to append messages to AgentStatus. You'll append other messages to the `AgentStatus` string in the following sections.

You haven't yet defined the `AgentStatus` property used in the previous code listing. Add a new automatic property to App.xaml.cs named `AgentStatus`:

```
public string AgentStatus { get; private set; }
```

In order to display the agent status on the screen, you add another `TextBlock` to MainPage.xaml. You place the `TextBlock` inside a `Border` and use the contrast background and styles to make the status messages stand out:

```
<Border Grid.Row="0"
    Background="{StaticResource PhoneContrastBackgroundBrush}">
    <TextBlock x:Name="agentMessage" TextWrapping="Wrap"
    Style="{StaticResource PhoneTextContrastStyle}"  />
</Border>
```

Next add a line to MainPage's `OnNavigatedTo` method to assign the `TextBlock`'s `Text` property:

```
agentMessage.Text = ((App)Application.Current).AgentStatus;
```

No message is displayed to the user when the periodic task is successfully created. Once created, the task will continue to execute twice an hour until its `ExpirationTime`. Unlike alarms and reminders, a scheduled task expires after two weeks.

3.6.4 Scheduled tasks expire after two weeks

By default, tasks are created with a maximum expiration time 14 days in the future. The developer can schedule a task with a shorter expiration time. Any attempt to specify an expiration time more than two weeks in the future will result in an exception. The two week limit requires the application to continuously reschedule the task to ensure that it'll be running as expected. This limitation also ensures that unused applications aren't draining device resources.

A good practice is to reschedule the task every time the application is launched. Update the `Application_Launching` method so that it looks for and removes any previously scheduled periodic task. Add the following snippet to the `Application_Launching` method, right before the `try/catch` block:

```
cleanupTask = ScheduledActionService.
    Find("NotificationCleanupTask") as PeriodicTask;
if (cleanupTask != null)
{
    if (cleanupTask.ExpirationTime < DateTime.Now)
        AgentStatus += "The background task was expired. ";

    ScheduledActionService.Remove(cleanupTask.Name);
}
```

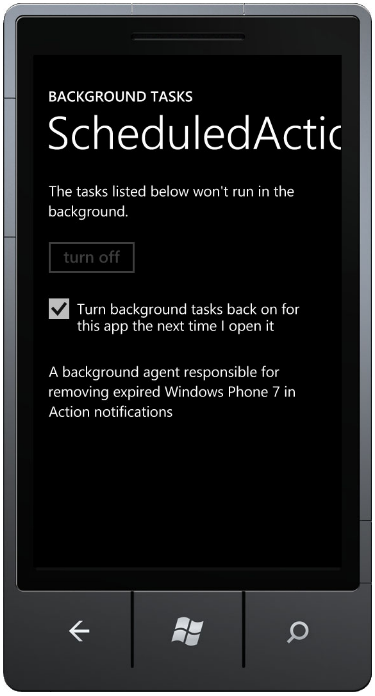
Before removing the previously scheduled task, the snippet checks the expiration time. If the expiration time is in the past, the user is notified that the existing task was expired.

Two week expiration dates aren't the only thing that prevents a periodic task from executing on schedule. The user can choose to disable your application's periodic task.

3.6.5 User-disabled tasks

`PeriodicTasks` can be disabled by the user in the settings application. Figure 3.9 shows a screenshot of the background tasks page of the settings application. If you examine the figure, you'll see how the text assigned to the `Description` property is shown to the user. In the screen shot, the user has disabled the periodic task, but will allow the application to recreate the task the next time it runs.

When the user has checked the Turn Back On option, the application will be allowed to remove and recreate the periodic task. If the option isn't checked, the Scheduled Action Service will throw an exception when attempting to recreate the task. An application can detect whether the user has disabled a task using the `IsEnabled` property. Add a check to the `Application_Launching` method to set the `AgentStatus` if the task has been disabled. The new code should be added inside the statement checking for null:


BACKGROUND TASKS

ScheduledActic

The tasks listed below won't run in the background.

☑ Turn background tasks back on for this app the next time I open it

A background agent responsible for removing expired Windows Phone 7 in Action notifications

Figure 3.9 The background tasks page of the settings application displays the periodic task's description and allows a user to disable the task.

```
if (cleanupTask != null)
{
    if (!cleanupTask.IsEnabled)
        AgentStatus += "The background task was disabled by the user. ";
...
}
```

Only periodic tasks can be disabled. Resource-intensive tasks are listed in the Advanced page of the background tasks settings application. The user can see the list of resource-intensive tasks, but can't disable them.

3.6.6 *When things go awry*

Earlier in the chapter you learned that a background agent notifies the operating system that work was successfully completed using the `NotifyComplete` method. In a perfect world, the scheduled agent would always succeed, but we don't live in a perfect world. When the background agent fails to complete its work, it should call the `Abort` method.

The `Abort` method informs the scheduled action service that the agent is unable to continue working. When the agent aborts, the Scheduled Action Service will set the `IsScheduled` property of the associated `ScheduledTask` to `false`, and will cease to trigger the agent.

Other scenarios will also cause the background agent to fail. Background agents are limited to 5 MB of memory, and when a background process exceeds the memory quota it's terminated. Periodic tasks are limited to 15 seconds of execution time and are terminated if the `NotifyComplete` method isn't called before the time limit is reached. An unhandled exception will also terminate the background agent. The exit status of the background agent is reported by the `LastExitReason` property of the scheduled task. The `LastExitReason` property returns one of the values in the `AgentExitReason` enumeration. The possible agent exit reasons are listed in table 3.3.

Table 3.3 Values of the `AgentExitReason` enumeration

Name	Description
`Aborted`	The background agent called the `Abort` method. The scheduled task's `IsScheduled` property has been set to false.
`Completed`	The background agent called the `NotifyComplete` method.
`ExecutionTimeExceeded`	The background agent failed to call `NotifyComplete` before its allocated time limit expired. Periodic tasks have a time limit of 15 seconds.
`MemoryQuotaExceeded`	The background agent attempted to allocate more than 5 MB of memory.
`None`	The background agent has not run.
`Other`	An unknown error occurred.
`Terminated`	The background agent was terminated early by the operating system due to conditions unrelated to the agent.
`UnhandledException`	The background agent failed to handle an exception produced during execution.

The host application should examine the `LastExitReason` property of its periodic tasks, and at a minimum report the error condition to the user. You're going to add code to the sample application to check the `LastExitReason` property and report any error conditions to the user. The sample application reports agent errors through the

`AgentStatus` property. Add the following snippet to the `Application_Launching` method, inside the check for `null` statement:

```
if (cleanupTask.LastExitReason != AgentExitReason.Completed
    && cleanupTask.LastExitReason != AgentExitReason.None)
{
    AgentStatus += string.Format("The background task failed to complete
➥ its last execution at {0:g} with an exit reason of {1}. ",
        cleanupTask.LastScheduledTime, cleanupTask.LastExitReason);
}
```

You ignore the `Completed` and `None` exit reasons. All other exit reasons result in a message that reports the exit reason and the last time the background agent was scheduled to run.

The notification sample is nearly complete. Each time the application starts up, the `PeriodicTask` that triggers the background agent is renewed, and its `ExpirationTime` is reset to a full two weeks. The application also handles various error conditions that may occur, displaying status information to the user. At this point you should launch the application in different circumstances and get a feel for how the background agent might be affected. Disable the background task, either with or without the Turn Back On option checked. What status messages do you see when you restart the application?

Unless you're really patient and waited half an hour or more, you probably haven't seen the background agent execute. Fortunately, Microsoft has provided an API to trigger early execution of background agents to enable developers to test.

3.6.7 Testing background agents

In normal situations a periodic task is only called every 30 minutes. Resource-intensive tasks are scheduled with even less predictability. Can you imagine having to wait half an hour for your background agent to trigger before you could debug and step through your code? The `ScheduledActionService` class provides the `LaunchForTest` API to give the developer control over when background agent execution is triggered.

The `LaunchForTest` method accepts the name of the `ScheduledTask`, and a `TimeSpan` value describing how soon the task should be triggered. The `LaunchForTest` method can be used by either the host application or by the scheduled agent. Add a call to `LaunchForTest` to the end of the `Application_Launching` method of the sample application:

```
if (cleanupTask != null)
    ScheduledActionService.LaunchForTest(cleanupTask.Name,
       TimeSpan.FromSeconds(3));
```

Debug the application again and a breakpoint in the `OnInvoke` should be hit within a few seconds.

`LaunchForTest` will only execute when an application is deployed to a device using the Windows Phone SDK tools. When debugging your project with a background agent, the debugger will continue to run even after the application has stopped. This is to allow your agent to be debugged even when the application isn't running.

3.7 Summary

Throughout this chapter we've covered features of the Windows Phone that allow applications to appear to be continuously running when they are not. A continuously running application can pose a problem for a mobile device with limited resources. A running application might interfere with the power management routines of the operating system and rapidly drain the battery. Runaway background applications steal processing power from foreground applications, resulting in a slow and unresponsive user experience.

Microsoft has designed the Windows Phone to provide the best user experience possible. The Windows Phone imposes limitations to ensure that applications can't intentionally or accidentally create performance or power problems for the user. One of these limitations prevents an application from running when not the foreground application.

In the first half of this chapter you learned how to detect when your application is switched from the foreground to the background. The Lifetime sample application demonstrated the various events raised when an application is launched, deactivated, activated, or closed. We showed you how and when to save application state and quickly recover when your application is switched back to the foreground.

In the second half of the chapter you learned how to use the Scheduled Action Service to implement features normally found in background applications. The Scheduled Action Service allows an application to schedule alarms and reminders, notifying the user at important points in time, and providing a quick link back into the application. The Scheduled Action Service is also used to schedule work with periodic and resource-intensive tasks. Scheduled tasks are used to trigger an application's background agent to perform work even when the application isn't running in the foreground.

Another limitation of the operating system prevents applications from directly accessing features in native applications such as the phone dialer, email, calendar, and contact database. In the next chapter you'll learn how to use launchers and choosers to integrate with the native applications. You'll also learn how to use the User Data API to read from the calendar and the contacts database.

4

Launching tasks and choosers

This chapter covers

- Using the phone APIs
- Launchers and choosers
- Searching for contact data
- Reading calendar appointments

A modern mobile phone does more than make phone calls. A mobile phone allows you to send SMS text messages and emails. Phone numbers and email addresses are stored in the phone's contact list. Appointments and meetings are viewed using a calendar application. Music and videos are played from the phone's media library. Don't forget about the ever-present camera. Windows Phone developers access these mobile phone features via the Tasks and UserData APIs.

The Windows Phone security model doesn't allow third-party applications to directly access the native applications and data stores provided by the operating system. Access to native applications is exposed through a variety of classes available in the Tasks API. Access to native data stores, specifically the contacts and calendar data stores, is exposed through the `Contacts` and `Appointments` classes found in the UserData API.

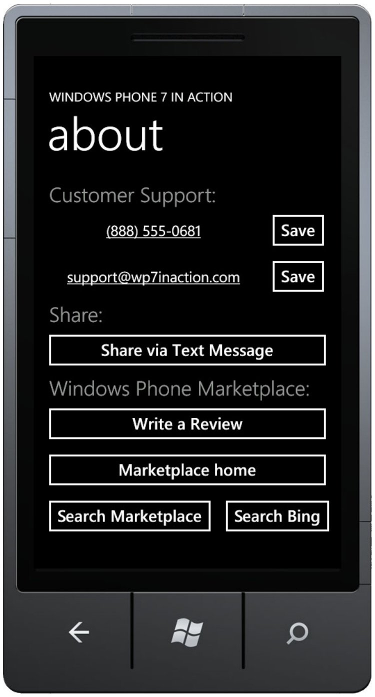
In this chapter we explore the UserData and Tasks APIs. You'll learn how to read contact and appointment data stored in the phone's People Hub and Calendar application. You'll also use tasks to access the native phone applications. You'll build two applications to demonstrate how to use the UserData and Tasks APIs. The sample you'll build at the end of the chapter will search for contacts and appointments. First you'll build an application that uses several different Tasks to initiate phone calls, emails, and text messages.

4.1 Tasks API

Phone tasks allow your code to interact with the native or built-in applications—phone dialer, media player, messaging, contacts, web browser, camera, and Marketplace. You're going to build an application, named PhoneTasks, with a sample About page that uses Tasks to contact customer support, share news about the application, add application reviews, and purchase a trial application.

> **NOTE** The *Application Certification Requirements for Windows Phone* specifies that every application provide easily discoverable technical support contact information.

To begin the PhoneTasks application project, create a new Windows Phone Application. The final PhoneTasks application is shown in figure 4.1. The user interface is built using hyperlinks and buttons. As you implement the application you'll show or launch the appropriate phone task from click event handlers in the page code behind.


WINDOWS PHONE 7 IN ACTION

about

Customer Support:

(888) 555-0681 Save

support@wp7inaction.com Save

Share:

Share via Text Message

Windows Phone Marketplace:

Write a Review

Marketplace home

Search Marketplace Search Bing

Figure 4.1 Screenshot of the PhoneTasks application with hyperlinks and buttons for launching native applications

The `ContentPanel` markup for the PhoneTasks project's main page is shown in the following listing. Open MainPage.xaml and copy the XAML markup into your project. As you add the various `Button` and `HyperlinkButton` controls, create corresponding empty `Click` event handlers in MainPage.xaml.cs.

Listing 4.1 Markup for the PhoneTask application's MainPage

```
<StackPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
   <TextBlock Text="Customer Support:"
      Style="{StaticResource PhoneTextGroupHeaderStyle}" />
   <StackPanel Orientation="Horizontal">
      <HyperlinkButton x:Name="supportPhoneLink" Width="325"
         Content="(888) 555-0681" Click="SupportPhoneLink_Click"
         Margin="{StaticResource PhoneTouchTargetOverhang}" />
      <Button x:Name="SavePhoneButton" Content="Save"
         Click="SavePhone_Click" />
```

① Change Content-Panel to a StackPanel

② Style labels as header text

```
   </StackPanel>
   <StackPanel Orientation="Horizontal">
      <HyperlinkButton x:Name="supportEmailLink" Width="325"
         Content="support@wp7inaction.com" Click="SupportEmailLink_Click"
         Margin="{StaticResource PhoneTouchTargetOverhang}" />    ❸ Leave enough room for touches
      <Button x:Name="saveEmailButton" Content="Save"
         Click="SaveEmail_Click" />
   </StackPanel>

   <TextBlock Text="Share:"
      Style="{StaticResource PhoneTextGroupHeaderStyle}" />
   <Button Content="Share via Text Message" Click="ShareSms_Click" />

   <TextBlock Text="Windows Phone Marketplace:"
      Style="{StaticResource PhoneTextGroupHeaderStyle}" />
   <Button Content="Write a Review" Click="Review_Click" />
   <Button x:Name="homePageButton" Content="Buy this application"
      Click="HomePage_Click" />
   <StackPanel Orientation="Horizontal">
      <Button Content="Search Marketplace" Click="Search_Click" />
      <Button Content="Search Bing" Click="BingSearch_Click" />
   </StackPanel>
</StackPanel>
```

You start by changing the `ContentPanel` from a `Grid` into a `StackPanel` ❶. Next you add the first of three header labels by creating a `TextBlock` and styling it with the theme resource named `PhoneTextGroupHeaderStyle` ❷. You use the customer support email address and phone numbers as the `Content` for two `HyperlinkButtons`. You use the theme resource named `PhoneTouchTargetOverhang` ❸ to ensure that each hyperlink has enough space around it to accommodate the thickness of a fingertip. Finally you add several `Buttons`, which the user will use to initiate one of the phone tasks. Run the application now and confirm the user interface appears as you'd expect.

The phone tasks are found in the `Microsoft.Phone.Tasks` namespace of the `Microsoft.Phone` assembly. Tasks don't share a common base class or interface, but every task implements a `Show` method. When the `Show` method is called, your application is deactivated and possibly terminated. Tasks are launched with the `Show` method because they're only shown. Your application can't use tasks to actually do anything. The tasks require the user to initiate all actions. Make sure to perform any necessary work before calling the `Show` method, especially if you're calling `Show` from the user interface thread. If you block or otherwise tie up the UI thread, the `Deactivated` event will queue up until the UI thread is free. If the user returns to your application before `Deactivated` is fired, your application will appear to hang.

Tasks come in two forms—launchers and choosers. Launchers are fire-and-forget; they show the task and don't return any data. Choosers show the task and return data to your application when it's reactivated.

4.2 Launchers

Launchers are a category of tasks that allow your code to activate a native or built-in application. Table 4.1 describes each of the launcher classes provided in the `Microsoft.Phone.Tasks` namespace. Data is passed to the launched application via properties set on the task. When the launcher's `Show` method is called, your application is deactivated.

Table 4.1 Windows Phone launchers

Launcher	Description
`BingMapsDirectionsTask*`	Launch the Bing Maps application showing driving directions between two locations
`BingMapsTask*`	Launch the Bing Maps application centered on a location.
`ConnectionSettingsTask`	Launch the Airplane Mode, Bluetooth, Cellular, or Wi-FI page of the settings application
`EmailComposeTask`	Launch the email application and send an email
`MarketplaceDetailTask`	Launch the specified application's home page in the Marketplace Hub
`MarketplaceHubTask`	Launch the Marketplace Hub
`MarketplaceReviewTask`	Launch an application's review page in the Marketplace Hub
`MarketplaceSearchTask`	Launch the search page in the Marketplace, and search for the specified keywords
`MediaPlayerLauncher`	Launch the media player and play the specified media
`PhoneCallTask`	Launch the phone dialer with the specified phone number and place a phone call
`SearchTask`	Launch the Bing application with the specified query
`ShareLinkTask`	Launch the Post a Link page in Internet Explorer
`ShareStatusTask`	Launch the Post a Message page in the People Hub
`SmsComposeTask`	Launch the messaging application and send a text message
`WebBrowserTask*`	Launch Internet Explorer with the specified web address

* BingMapsDirectionsTask, BingMapsTask, and WebBrowserTask are covered in chapter 13.

`PhoneCallTask`, `EmailComposeTask`, and `SmsComposeTask` are contact tasks. You'll use these tasks if your application is some form of contact management application or if any of your data contains phone numbers or email addresses. For this example you're going to use them in the PhoneTasks application to provide links to customer support.

`ShareLinkTask` and `ShareStatusTask` are social networking tasks. You'll use these tasks to allow users of your application to post links and status messages to Windows Live, Facebook, Twitter, or LinkedIn. Using the social networking tasks is similar to using the other launchers and we don't cover them in this book. Neither of the social networking tasks work with the emulator because the emulator doesn't allow you to set up email and social networking accounts. You can see an example of how to use them in the PhoneTasks application that's available with the book's sample source code, which is slightly different than the application you build in this chapter.

The marketplace tasks provide integration with the Marketplace Hub. You're going to use them in your PhoneTasks application—giving users quick access to the review and buy marketplace pages, and to search for other applications in the portfolio.

You're going to start the PhoneTasks application implementation by using `PhoneCallTask` to call customer support.

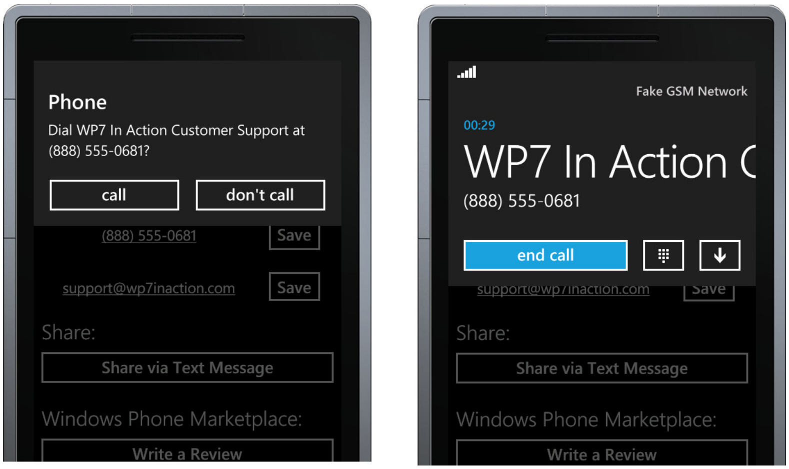
4.2.1 Placing a phone call

`PhoneCallTask` shows the phone dialer, prompting the user to dial a phone number specified by the application. The user must initiate the phone call; the application can't actually dial a phone call directly. `PhoneCallTask` has a `PhoneNumber` property and a `DisplayName` property.

> **TIP** Use of the `PhoneCallTask` requires the `ID_CAP_PHONEDIALER` capability to be declared in WMAppManifest.xml

Figure 4.2 shows how the user is prompted when `PhoneCallTask.Show` is executed. The dialer screen overlays the top portion of the application, while the remainder of the screen is disabled. Unlike all other tasks, `PhoneCallTask` doesn't pause or terminate the application, but only obscures it. The application is notified via the `PhoneApplicationFrame.Obscured` event. If the user clicks Call, the in-call screen overlays the running application.

The in-call overlay presents the `DisplayName` if the phone number doesn't match an existing contact. When a phone number matches an existing contact, the data in



Phone

Dial WP7 In Action Customer Support at (888) 555-0681?

call
don't call
(888) 555-0681
Save
support@wp7inaction.com
Save
Share:
Share via Text Message
Windows Phone Marketplace:
Write a Review
Fake GSM Network
00:29
WP7 In Action C
(888) 555-0681
end call
support@wp7inaction.com
Save
Share:
Share via Text Message
Windows Phone Marketplace:
Write a Review

Figure 4.2 Phone dialer and in-call overlays

the contact record is shown instead of the specified `DisplayName`. The user might return to the running application by tapping below the in-call overlay, which generates a `PhoneApplicationFrame.Unobscured` event. The user might also take another action that will result in the deactivation of your application. The following listing shows how to use `PhoneCallTask`.

Listing 4.2 Launching the phone dialer with the `PhoneCallTask`

```
using Microsoft.Phone.Tasks;

private void SupportPhoneLink_Click(object sender, RoutedEventArgs e)
{
    PhoneCallTask task = new PhoneCallTask()
    {
        PhoneNumber = (string)SupportPhoneLink.Content,
        DisplayName = "WP7 In Action Customer Support"
    };
    task.Show();
}
```

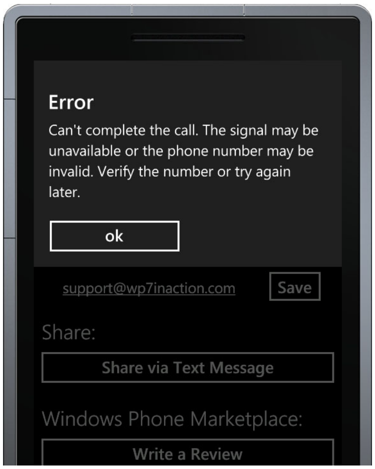
① **Declare namespace**

② **Construct and initialize the task**

The `PhoneCallTask` class is defined in `Microsoft.Phone.Tasks`, so you add a using directive ① for the namespace at the top of the file. Next, implement the `SupportPhoneLink_Click` event handler by constructing a `PhoneCallTask` ② with object initializers for the `PhoneNumber` and `DisplayName` properties. Finally the `Show` method is called to prompt the user to place the phone call.

The `PhoneNumber` property is required. If `PhoneNumber` isn't set, `PhoneCallTask.Show` will return immediately without performing any work, and the dialer screen won't be displayed to the user. The user's only options are to dial or not to dial and the user can't change the phone number. The `DisplayName` property isn't required, and if not specified, the in-call screen will only display the phone number. When the `PhoneNumber` property isn't a valid phone number, an error message will be displayed to the user after they press the Call button as shown in figure 4.3.

Some users will prefer to interact with customer support via email instead of a phone call. For this example you'll use `EmailComposeTask` to allow these users to send an email.



Error

Can't complete the call. The signal may be unavailable or the phone number may be invalid. Verify the number or try again later.

Figure 4.3 Error message displayed when an invalid phone number is specified

4.2.2 *Writing an email*

The `EmailComposeTask` shows the email application, prompting the user to send an email specified by the application to a specified email address. The user must initiate the send; your application can't actually send an email message directly. `EmailComposeTask` has `To`, `CC`, `Subject`, and `Body` properties. The next listing shows how to show the `EmailComposeTask`.

Listing 4.3 Composing an email with the `EmailComposeTask`

```
private void SupportEmailLink_Click(object sender, RoutedEventArgs e)
{
    EmailComposeTask task = new EmailComposeTask()
    {
        To = (string)SupportEmailLink.Content.ToString(),
        Subject = "WP7 in Action PhoneTasks Application",
        Body = "Support Issue Details:"
    };
    task.Show();
}
```

① **Construct and initialize task**

Implement the `SupportEmailLink_Click` event handler by constructing an `EmailComposeTask` ① with object initializers for the `To`, `Subject`, and `Body` properties. Next, the `Show` method is called to launch the email editor.

NOTE The email application isn't accessible in the emulator, and when the `EmailComposeTask` is shown on the emulator, the user is shown an error message.

All of the properties are optional. The messaging application will launch with a new empty email when nothing is supplied. The `To` and `CC` properties can be one or more names or email addresses, separated with a semicolon. The messaging application will attempt to match the `To` and `CC` values to contacts in the address book after the user presses send. Once in the email editor, the user can alter, delete, or replace any of the values passed in from your application.

The last contacts-related launcher we're going to look at is the `SmsComposeTask`.

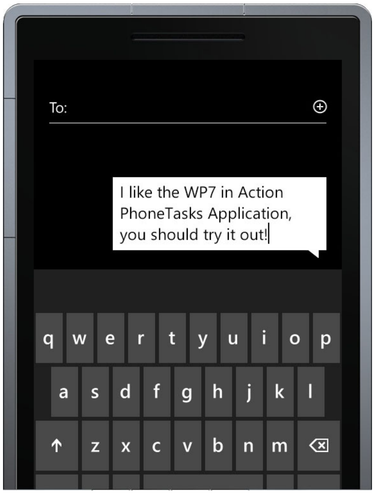
4.2.3 Texting with SMS

The `SmsComposeTask` shows the text messaging application, prompting the user to send a text specified by your application to a specified name or phone number. The user must initiate the send, as your application can't actually send a text message directly. `SmsComposeTask` has `To` and `Body` properties.

You're going to use the `SmsComposeTask` to provide the user a means to send information and news about the application to contacts in their address book, a form of high-tech word-of-mouth advertising.

Figure 4.4 shows the messaging application as the user sees it when `SmsComposeTask.Show` is executed.

A `SmsComposeTask` is constructed in the `ShareSms_Click` event handler with object initializers for the `Body` property. The `Show` method is called to launch the messaging application:



I like the WP7 in Action PhoneTasks Application, you should try it out!

Figure 4.4 Messaging application

```
private void ShareSms_Click(object sender, RoutedEventArgs e)
{
    SmsComposeTask task = new SmsComposeTask()
    {
        Body = "I like the WP7 in Action
➥ PhoneTasks Application, you should try it out!",
    };
    task.Show();
}
```

Both the `To` and `Body` properties are optional. The messaging application will launch with a new empty message when neither are supplied. The `To` property can be one or more names or a phone number, separated with a semicolon. The messaging application will attempt to match the `To` values to contacts in the address book after the user presses Send. Once in the messaging editor, the user can alter, delete, or replace the `To` and `Body` values passed in from your application.

As with other tasks, the user can return to your application by pressing the Back button. The user can return without sending the text. The user may also send the text and then choose to perform other actions without ever returning to your application. With the contacts features of the About page implemented, you're ready to implement the marketplace tasks.

4.2.4 Working with the Marketplace

The Windows Phone Marketplace Hub, shown in figure 4.5, is where users go to download, review, and purchase applications for the Windows Phone. Three different tasks are available to launch the Marketplace Hub from third-party applications.

The `MarketplaceHubTask` launches the Marketplace Hub to either the applications or music portal. The other two tasks, `MarketplaceReviewTask` and `MarketplaceDetailTask`, will launch the Marketplace Hub directly to an application's site

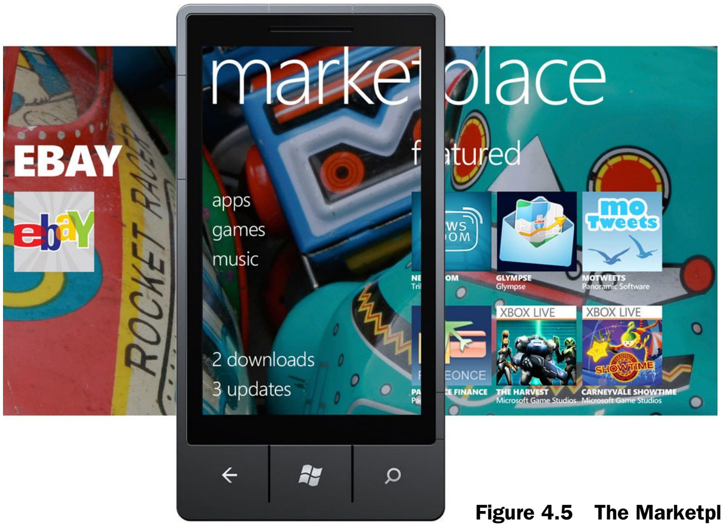


Figure 4.5 The Marketplace Hub

within the Marketplace. Let's take a closer look at each task, starting with `MarketplaceHubTask`.

MARKETPLACEHUBTASK

The first marketplace task we'll discuss is the `MarketplaceHubTask`. This task launches the Windows Phone Marketplace Hub. `MarketplaceHubTask` has a single `ContentType` property. Launch the task with the `Show` method:

```
MarketplaceHubTask task = new MarketplaceHubTask()
{
    ContentType = MarketplaceContentType.Applications
};
task.Show();
```

The `ContentType` property must be set to one of the `MarketplaceContentType` enumeration values—`Applications or Music`. `ContentType` defaults to `MarketplaceContentType.Applications`.

You don't have a need to launch the Marketplace Hub start pages. But you do want to launch the Marketplace's *Create a Review* feature. `MarketplaceReviewTask` allows you to do just that.

MARKETPLACEREVIEWTASK

Reviews are a big part of the Windows Phone Marketplace. The Windows Phone Marketplace will grow to tens or hundreds of thousands of applications. Reviews help the consumer find your great application in a sea of mediocre competitors. With the `MarketplaceReviewTask` Microsoft has made it easy to encourage users to create reviews for your application.

To wire up the review feature to the About page, you'll implement the `Click` handler for the Review button:

```
private void Review_Click(object sender, RoutedEventArgs e)
{
    MarketplaceReviewTask task = new MarketplaceReviewTask();
    task.Show();
}
```

`MarketplaceReviewTask` doesn't have any public properties. The task determines the appropriate product ID from the running application, so when you call `Show` the right page in the Marketplace is shown.

Offering more than the review page, the Marketplace allows you to direct the user to an application's home page with the `MarketplaceDetailTask`.

MARKETPLACEDETAILTASK

An application's home page in the Windows Phone Marketplace provides all the details, screenshots, and reviews for an application. It's also where a user goes to purchase an application or download a trial. In your About page, you'll offer a link to your application's Marketplace home page. When the application runs under a trial license, you want to show the user a Buy Now button as shown in figure 4.6.

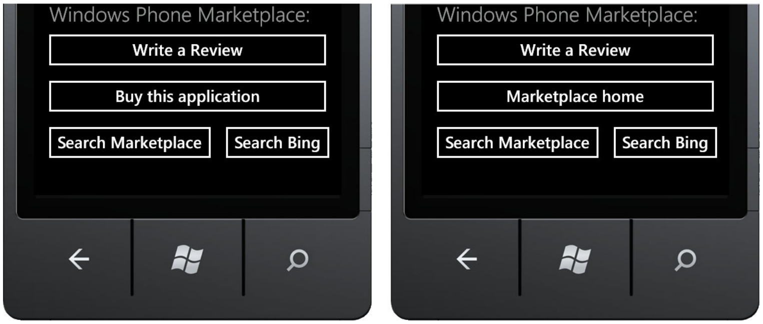


Figure 4.6 The home page button's content in trial and licensed modes

You learned how to use the `IsTrial` API to change application behavior in chapter 2. The next listing uses the `IsTrial` API to update the home page button's text.

Listing 4.4 Using `IsTrial` to customize the HomeButton's text

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    LicenseInformation licenseInfo = new LicenseInformation();
    if (licenseInfo.IsTrial())
        HomePageButton.Content = "Buy this application";
    else
        HomePageButton.Content = "Marketplace home";

    base.OnNavigatedTo(e);
}
```

❶ **Is app using trial license?**

❷ **Update button's content**

You're adding code to the `OnNavigatedTo` override, because it's possible for the application to be licensed while it's deactivated. You start by creating an instance of `LicenseInformation` and checking for a trial license ❶. When running under a trial license, the button text encourages the user to buy the application ❷.

Next you need to implement the `HomePage_Click` event handler:

```
private void HomePage_Click(object sender, RoutedEventArgs e)
{
    MarketplaceDetailTask task = new MarketplaceDetailTask();
    task.Show();
}
```

`MarketplaceDetailTask` has two properties—`ContentIdentifier` and `ContentType`. The `ContentType` property defaults to `MarketplaceContentType.Applications`, which is the only valid value for this task. `ContentIdentifier` defaults to `null`, which indicates that the task should use the product identifier of the currently running application.

`MarketplaceDetailTask` also allows you to display the home page for any application, if you know its `ContentIdentifier`. Your application can provide links to all the applications published by you, as you're likely to know those IDs. Another option for showing the user all of your applications is with a Marketplace search.

MARKETPLACESEARCHTASK

Our About page utilizes the `MarketplaceSearchTask` to search for related applications. The search is defined with a set of search terms, which for this application are “Windows Phone 7 in Action”:

```
private void Search_Click(object sender, RoutedEventArgs e)
{
    MarketplaceSearchTask task = new MarketplaceSearchTask()
    {
        SearchTerms = "Window Phone 7 in Action",
        ContentType = MarketplaceContentType.Applications
    };
    task.Show();
}
```

Implement the `Search_Click` event handler by constructing a `MarketplaceSearchTask` with object initializers for the `SearchTerms` and `ContentType` properties. The `Show` method is called to launch the marketplace.

Searching with Bing is an alternative to searching the Application Marketplace. The Tasks API includes a task to launch a Bing Search.

4.2.5 Searching with Bing

The last task you’ll use in the PhoneTasks sample application is `SearchTask`. `SearchTask` launches the Bing Search application with a specified search query. The PhoneTasks application utilizes the `SearchTask` to search for “Windows Phone 7 in Action” on Manning’s website:

```
private void BingSearch_Click(object sender, RoutedEventArgs e)
{
    SearchTask task = new SearchTask()
    {
        SearchQuery = "Windows Phone 7 in Action site:manning.com"
    };
    task.Show();
}
```

The search query is constructed using the `site` operator defined in the Bing Query Language. The Bing Query Language is documented on MSDN at http://mng.bz/6NXP.

If the `SearchQuery` property is a null or empty string, the `Show` method returns immediately and doesn’t launch the Bing Search application.

The `PhoneTasks` application is nearly finished. The remaining buttons allow the user to save the support phone number and email address to the contacts database. Saving data to contacts requires the user to choose which contact to update. For this you need choosers.

4.3 Choosers

Choosers are a category of Windows Phone tasks that return task status and data to an application. Each chooser displays a native application user interface and either saves data to the native application or returns data from the native application. Table 4.2 describes the choosers available in the `Microsoft.Phone.Tasks` namespace. Because applications might be terminated when a task is shown, there's no guarantee the user will ever return to your application with the chosen data.

Table 4.2 Windows Phone choosers

Chooser	Description
`AddressChooserTask`	Launch the 'choose a contact page' in the People Hub and return a physical address of the user's choosing
`CameraCaptureTask*`	Launch the camera application and return a photo the user takes
`EmailAddressChooserTask`	Launch the 'choose a contact page' in the People Hub and return an email address of the user's choosing
`PhoneNumberChooserTask`	Launch the 'choose a contact page' in the People Hub and return a phone number of the user's choosing
`PhotoChooserTask*`	Launch the Pictures application and return a picture of the user's choosing
`SaveContactTask`	Launch the 'new contact' page in the People Hub prompting the user to edit and save a new contact
`SaveEmailAddressTask`	Launch the 'choose a contact' page in the People Hub prompting the user to add an email address to the contact of the user's choosing
`SavePhoneNumberTask`	Launch the 'choose a contact' page in the People Hub prompting the user to add a phone number to the contact of the user's choosing
`SaveRingtoneTask`	Launch the Ringtones application prompting the user to save an audio file from the application's storage as a ringtone

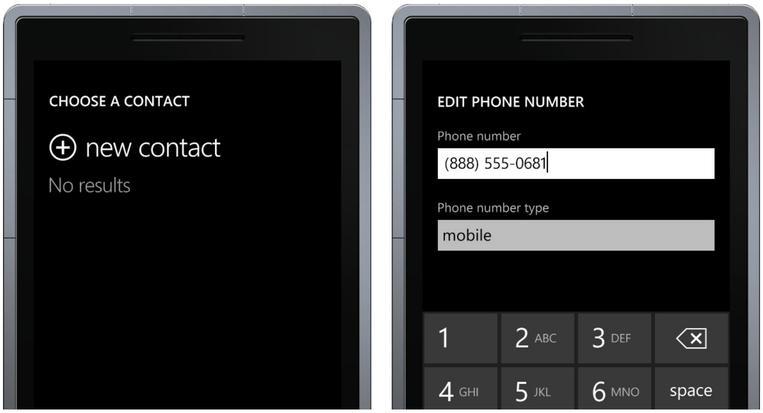
`*CameraCaptureTask` and `PhotoChooserTask` are covered in chapter 6.

`AddressChooserTask`, `EmailAddressChooserTask`, and `PhoneNumberChooserTask` return data chosen by the user. `SaveContactTask`, `SaveEmailAddressTask`, `SavePhoneNumberTask`, and `SaveRingtoneTask` don't return any data; they only return status information. Data and status information are returned to your application via the `TaskEventArgs` parameter of the chooser's `Completed` event handler.

4.3.1 Completed events

Choosers, like all tasks, are launched with the `Show` method. When `Show` executes, your application is deactivated and possibly tombstoned. Upon activation, the Windows Phone framework raises the chooser's `Completed` event. Because the completed event is called when a tombstoned application is restarted, `Completed` events should be treated specially.

Chooser instances should be declared as instance fields of the `PhoneApplicationPage`. To enable callbacks when recovering from tombstoning, `Completed` event handlers must be wired up in the Silverlight `Page` constructor or `Loaded` event handler. When



CHOOSE A CONTACT

new contact

No results

EDIT PHONE NUMBER

Phone number

(888) 555-0681

Phone number type

mobile

Figure 4.7 Saving an application-supplied phone number

the event handler is added in a page constructor, the event handler is called immediately after the constructor, before the `Loaded` event. But if the event handler is added in the `Loaded` event, the `Completed` event is called immediately after the `Loaded` event.

You'll use `SavePhoneNumberTask` and `SaveEmailAddressTask` to finish the PhoneTasks application.

4.3.2 Saving a phone number

One feature of the PhoneTasks application allows the user to save the customer support phone number to the contacts database. Once the phone number is saved, the feature is hidden. The feature will be implemented with `SavePhoneNumberTask`. When the chooser launches, the user is prompted to select a new or existing contact, and is then presented with the Edit Phone Number screen shown in figure 4.7.

At this point, the user can change the phone number or return to the application without saving the number (or start a completely different application, never returning to PhoneTasks). The following listing shows the code added to MainPage.xaml.cs for creating an instance field and adding the completed event handler in the constructor.

Listing 4.5 Wiring up the completed event handler

```
SavePhoneNumberTask savePhoneNumberTask = new SavePhoneNumberTask();

public MainPage()
{
    InitializeComponent();
    savePhoneNumberTask.Completed += savePhoneTask_Completed;
}

private void savePhoneTask_Completed(object sender, TaskEventArgs e)
{
    if (e.TaskResult == TaskResult.OK)
        SavePhoneButton.Visibility = Visibility.Collapsed;
}
```

Add field for task

Add completed event handler

① **Confirm save and hide button**

You only want to hide the `SavePhoneButton` when the returned status is `TaskResult.OK` ❶. You hide the button by changing its `Visibility` property to `Visibility.Collapsed`.

Finally you can implement the `SavePhone_Click` event handler. The phone number you're asking the user to save comes from the `SupportPhoneLink` control:

```
private void SavePhone_Click(object sender, RoutedEventArgs e)
{
    savePhoneNumberTask.PhoneNumber = (string)SupportPhoneLink.Content;
    savePhoneNumberTask.Show();
}
```

Saving the customer support email address is the only remaining feature left to implement. You'll use `SaveEmailAddressTask` to complete PhoneTasks.

4.3.3 Saving an email address

The last feature of the PhoneTasks application allows the user to save the customer support email address to the contacts database. As with the phone number, you'll hide the feature when the email address is saved. When the `SavePhoneNumberTask` chooser launches, the user is prompted to select a new or existing contact, and is then shown the edit email address screen in figure 4.8.

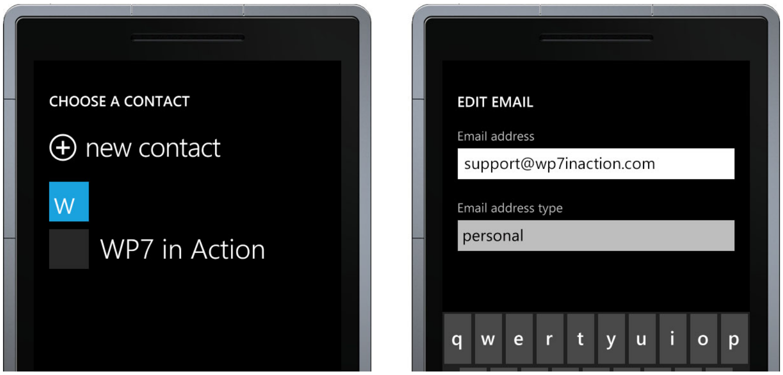
Because this is another chooser, you need to add an instance field to the class:

```
SaveEmailAddressTask saveEmailAddressTask = new SaveEmailAddressTask();
```

Next, you wire up the `Completed` event handler in the constructor and implement the event handler:

```
saveEmailAddressTask.Completed += saveEmailTask_Completed;
```

```
private void saveEmailTask_Completed(object sender, TaskEventArgs e)
{
    if (e.TaskResult == TaskResult.OK)
        SaveEmailButton.Visibility = Visibility.Collapsed;
}
```



Email address
support@wp7inaction.com

Email address type
personal

Figure 4.8 Saving an application supplied email address

Finally, you implement the `SaveEmail_Click` event handler. The email address you're asking the user to save comes from the `SupportEmailLink` control:

```
private void SaveEmail_Click(object sender, RoutedEventArgs e)
{
    saveEmailAddressTask.Email = (string)SupportEmailLink.Content;
    saveEmailAddressTask.Show();
}
```

You're done adding Windows Phone tasks to the PhoneTasks application, but not quite finished looking at chooser tasks. Before we move on to the next sample application, we discuss saving ringtones and choosing physical addresses, phone numbers, and email addresses.

4.3.4 Saving a ringtone

A *ringtone* is the sound played when a phone receives an incoming call, voice mail, text message, or email message. Ringtones are implemented with regular audio files that are installed on the phone. Windows Phone supports ringtones that are either MP3 or WMA audio files. To qualify as a ringtone, the audio file must be less than 40 seconds long and smaller than 1 MB. Ringtones must not have digital rights management (DRM) protection.

> **NOTE** A ringtone-saving application is included with the books sample code, which is available from the book's website at http://www.manning.com/perga.

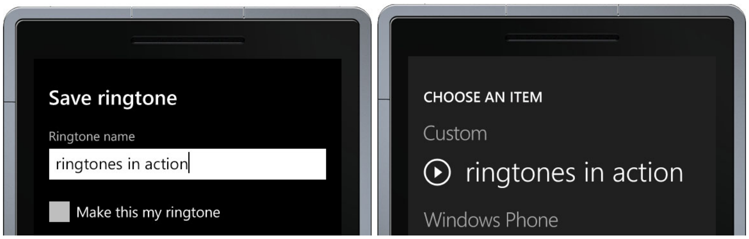
The `SaveRingtoneTask` enables the development of applications that install custom ringtones. Custom ringtone audio files are read from the application install folder or isolated storage, which means your application must either include the ringtones in its .xap file, or download them into isolated storage before installing them with `SaveRingtoneTask`. The next listing demonstrates how to install a custom ringtone.

Listing 4.6 Saving a ringtone with `SaveRingtoneTask`

```
private void SaveRingtone_Click(object sender, EventArgs e)
{
    SaveRingtoneTask task = new SaveRingtoneTask()
    {
        DisplayName = "ringtones in action",
        IsoStorePath = new Uri("isostore:/transfers/ringtone.wma"),
    };
    task.Completed += saveRingtoneTask_Completed;
    task.Show();
}
```

Read audio file from isolated storage ❶

The `SaveRingtoneTask` is used like all the other choosers. The task is constructed, its properties are set, the `Completed` event is hooked, and the `Show` method is called. The `DisplayName` property is optional and, if not set, will default to the name of the audio file. The `IsoStorePath` property is required and must be an `Uri` using either the `appdata` or `isostore` schema. In this listing, you're referencing an audio file ❶ that was downloaded to isolated storage.



Save ringtone

Ringtone name

ringtones in action

☐ Make this my ringtone

CHOOSE AN ITEM

Custom

ringtones in action

Windows Phone

Figure 4.9 Saving a ringtone (left) and selecting a custom ringtone for a contact (right)

Once the chooser's user interface is displayed, the user can change the display name of the ringtone. The chooser UI is shown in figure 4.9. The chooser user interface will only be shown if the audio file specified in the `IsoStorePath` is found at the given path and is an MP3 or WMA file. If the audio file is longer than 40 seconds or larger than 1 MB, the `Error` property in the `TaskEventArgs` returned to the `Completed` event handler will hold a `FormatException`.

`SaveRingtoneTask`, along with `SaveContactTask`, `SavePhoneNumberTask`, and `SaveEmailAddressTask`, displays a native user interface and allows data from a third-party application to be sent to a native application. These three tasks return status information to the calling application. The next set of chooser tasks that you learn about send data from the native applications to third-party applications. Your application can use the choosers to obtain a phone number, email address, or street address.

4.3.5 Choosing a phone number

`PhoneNumberChooserTask` retrieves a phone number from the contacts database, prompting the user to choose a phone number with the built-in contacts user interface. The `PhoneNumberChooserTask.Completed` event uses the `PhoneNumberResult` event args. The following listing shows how to use `PhoneNumberResult`.

Listing 4.7 Retrieving the phone number from `PhoneNumberChooserTask`

```
void phoneChooser_Completed(object sender, PhoneNumberResult e)
{
    if (e.Error != null)
       chooserResult = e.Error.Message;
    else if (e.TaskResult == TaskResult.Cancel)
        chooserResult = "user canceled";
    else if (e.TaskResult == TaskResult.None)
        chooserResult = "no result";
    else if (e.TaskResult == TaskResult.OK)
        chooserResult = string.Format( "Phone Number for {0}\r\n{1}",
            e.DisplayName, e.PhoneNumber);
}
```

① **Check for exceptions**

② **User canceled**

③ **Use selected phone number**

In this snippet, `chooserResult` is a `string` field. `Completed` event handlers should check the `Error` property on the event args parameter ①. If `Error` isn't null, an exception occurred during the choose operation. When the user presses the back button to return to the application without selecting a phone number, a `TaskResult.Cancel` status is returned ❷ and the `PhoneNumber` property will be null. When the user completes the task, `TaskResult.OK` is returned, and the `PhoneNumber` property ❸ contains the selected value.

When your application requires an email address instead of a phone number, use `EmailAddressChooserTask`.

4.3.6 Choosing an email address

`EmailAddressChooserTask` retrieves an email address from the contacts database using the built-in contacts user interface. Your application depends on user interaction to choose the email address. The difference between `EmailAddressChooserTask` and `PhoneNumberChooserTask` lies in the type of result object passed to the `Completed` event handler. `EmailAddressChooserTask` uses `EmailResult` which provides an `Email` property:

```
if (e.TaskResult == TaskResult.OK)
    chooserResult = string.Format("Email Address for {0}\r\n{1}",
        e.DisplayName, e.Email);
```

The last chooser we explore is the `AddressChooserTask`.

4.3.7 Choosing a street address

A physical address can be retrieved from the contacts database using the `AddressChooserTask` task. Just like the other chooser tasks, the user is prompted to select an address with a built-in user interface. The `AddressChooserTask` returns an `AddressResult` object passed to the `Completed` event handler. The `Address` property of `AddressResult` provides the physical address in the form of a `string`:

```
if (e.TaskResult == TaskResult.OK)
    chooserResult.Text = string.Format("Street Address for {0}\r\n{1}",
        e.DisplayName, e.Address);
```

`AddressChooserTask` rounds out the list of choosers `discussed` in this chapter. In the next section you'll read directly from the contacts database to retrieve more than just a phone number, email address, or physical address. Even though using the contacts choosers is limited, you should consider using them instead of reading directly from the contacts database. The choosers use the built-in user interface, saving the developer the work required to build a custom UI. The choosers provide the same consistent UI that phone users see when working in the People Hub.

What are you to do if your application requires more than just phone numbers, email addresses, and physical addresses? The UserData API exposes nearly all of the data stored in the contact database.

4.4 UserData APIs

Windows Phone 7 aggregates contacts and calendar data from multiple service providers and social networks. Contacts from Windows Live, Facebook, and email accounts all appear in the People Hub. Appointments from each of the user's accounts are displayed in the Calendar application. The UserData API exposes a read-only view of the data that appears in the People Hub and the Calendar applications.

> **NOTE** Users will be notified that an application reads contacts and appointments data when they download the application from the Marketplace. The Marketplace doesn't tell them how an application will use the data. Applications that use the UserData API should respect the user's privacy and inform them how the data will be used. If the data is sent over the network, you should tell the user.

The UserData API consists of the classes and enumerations found in the `Microsoft.Phone.UserData` namespace. The two topmost classes are `Appointments` and `Contacts`, which provide the methods used to search through the calendar and contacts databases. Another important class is the `Account` class, which is used to identify the source of the data. Many of the classes in the UserData namespace have an `Account` property that identifies which service provided the contact or appointment data.

The `Account` class has two properties named `Name` and `Kind`. The `Name` property displays the name value entered by the user when the account was first created in the settings application. The `Kind` property is of type `StorageKind` and will have one of the enumeration values listed in table 4.3.

Table 4.3 The kind of accounts exposed by the UserData API

StorageKind value	Description
`Phone`	The contact or appointment was created on the phone and isn't associated to any service provider.
`Windows Live`	The contact or appointment data synchronized with a Windows Live account.
`Outlook`	The contact or appointment data synchronized with a Microsoft Outlook account.
`Facebook`	Contact data synchronized with Facebook.
`Other`	Contact and appointment data synchronized with some other service provider.

To demonstrate how to work with the UserData API you'll build a new sample application. Shown in figure 4.10, the application will contain a `Pivot` control with two `PivotItems`, one for searching contacts and the other for searching appointments. The `Pivot` control is analogous to a tab control, and we cover it in depth in chapter 10. Open Visual Studio, select New Project from the File menu, choose the Windows Phone Application project template, and name the project UserData. You're starting with the basic application instead of the Windows Phone Pivot Application template.

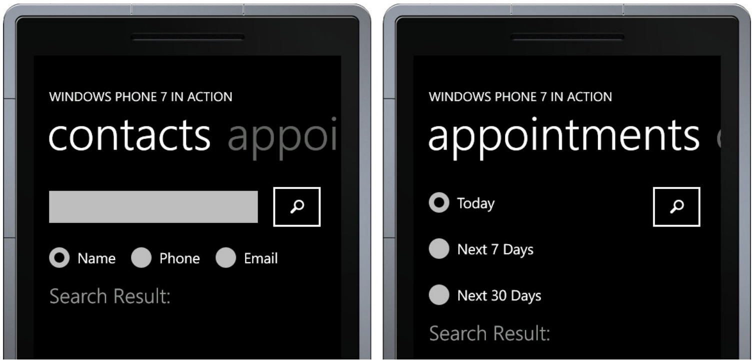


Figure 4.10 The user data sample application

You're not using the Pivot Application Template since you don't need sample Model-View-ViewModel code generated for you.

The MainPage.xaml file generated by the Windows Phone Application template isn't going to work here and you need to delete it from the project. Create a new MainPage.xaml using the Project > Add New Item menu option. From the New Item dialog, choose the Windows Phone Pivot Page item template and name the new page MainPage.xaml. If you look at the new MainPage.xaml file, you see a `Pivot` control containing two empty `PivotItems`. In the sample project, you'll add the controls shown in figure 4.10 to the `PivotItems`. You can read more about the `Pivot` control in chapter 10.

The first `PivotItem` contains controls to allow the user to search for a contact by name, phone number, or email address.

4.4.1 Searching for contacts

Contact records are discovered by executing a search against the contacts data store. The UserData API provides the `Contacts` class as the façade providing access to each of the service provider's contacts data. Five different types of searches can be performed. A search can return all the data available in the data store. Searches can be restricted to look for specific names, phone numbers, or email addresses. A search can also return the contacts that the user has pinned to the start screen.

> **TIP** Use of the Contacts API requires the `ID_CAP_CONTACTS` capability to be declared in WMAppManifest.xml.

Your sample application allows the user to enter search terms, specify a search type, and then execute the search. The contact search user interface is implemented in the contacts `PivotItem`. The XAML markup for contact search is shown next.

Listing 4.8 XAML markup for the contact pivot item

```
<controls:PivotItem Header="contacts">
    <StackPanel>
        <StackPanel Orientation="Horizontal">
```

❶ **Relabel first item**

```
            <TextBox x:Name="filterBox" Width="350" />
            <Button Click="searchContacts_Click">
                <Image Source="/Images/appbar.feature.search.rest.png" />
            </Button>
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <RadioButton x:Name="nameSearch" Content="Name"
                IsChecked="True" />
            <RadioButton x:Name="phoneSearch" Content="Phone" />
            <RadioButton x:Name="emailSearch" Content="Email" />
        </StackPanel>
        <TextBlock Text="Search Result:"
            Style="{StaticResource PhoneTextGroupHeaderStyle}" />
        <TextBlock x:Name="contactsResult"
            Margin="{StaticResource PhoneMargin}"/>
        <ContentControl x:Name="contactControl"
            Margin="{StaticResource PhoneMargin}">
            <ContentControl.ContentTemplate>
                <DataTemplate>
                </DataTemplate>
            </ContentControl.ContentTemplate>
        </ContentControl>
    </StackPanel>
</controls:PivotItem>
```

❷ **Radio buttons to specify search kind**

❸ **Display results**

You start the code listing by changing the header ❶ of the first `PivotItem` from *item1* to *contacts*. You add several controls to the `PivotItem` starting with a `TextBox` allowing the user to input the search term, and an image button to execute the search. The `Button`'s `Click` event is handled by a method named `searchContacts_Click` which you'll implement in the following pages. Three `RadioButton` controls are added ❷ to allow the user to specify the kind of search to execute—name, phone number, or email. Finally you add a few controls to display the results of the search ❸. The `contactControl`'s `ContentTemplate` property is an empty `DataTemplate` to start with and you'll be adding controls to display details about the resulting contact records.

Searches are executed using an asynchronous pattern with the `SearchAsync` method of the `Contacts` class. Results are returned via a completed event named `SearchCompleted`. `SearchAsync` accepts a search term and a filter type. Allowable values for the filter type are `None`, `PinnedToStart`, `EmailAddress`, `PhoneNumber`, and `DisplayName`. These values are defined by the `FilterKind` enumeration. Like many other asynchronous methods, `SearchAsync` also accepts a state object that calling code can use to pair method calls to completed events.

Before implementing the `searchContacts_Click` method, you need to declare and construct an instance of the `Contacts` class. Add a field to the `MainPage` class to hold the `Contacts` instance:

```
Contacts contacts = new Contacts();
```

With the contacts instance created you're ready to search the contacts data store. The next listing implements the `searchContacts_Click` method where you perform the search.

Listing 4.9 Searching for a contact

```
private void searchContacts_Click(object sender, RoutedEventArgs e)
{
    var filterKind = FilterKind.DisplayName;
    if (phoneSearch.IsChecked.Value)
        filterKind = FilterKind.PhoneNumber;
    else if (emailSearch.IsChecked.Value)
        filterKind = FilterKind.EmailAddress;

    string filter = filterBox.Text;

    contacts.SearchAsync(filter, filterKind, null);
}
```

❶ **Specify type of search**

❷ **Execute search**

First you determine the type of search to perform by looking at the `RadioButtons` ❶. If a radio button is checked, you use the corresponding `FilterKind` enumeration value. Next you retrieve the search term from the `TextBox` before you finally call the `SearchAsync` method ❷.

Phone number searches look for an exact match of all of the digits in the search term. Email searches match the search term to the user name or user name and domain name. Display name searches match the search term with the contact's first or last name using a *starts with* algorithm.

Your sample application doesn't use `None` or `PinnedToStart`, two of the five possible `FilterKind` values. Searches conducted with the `None` value will ignore the search term and return all contacts in the data store. The search term is also ignored by `PinnedToStart` searches, which return all of the contacts who have live tiles pinned to the start screen.

Search results are returned to the application via the `SearchCompleted` event. Event handlers are passed an instance of the `ContactsSearchEventArgs` class, which exposes properties containing the search term, the `FilterKind` value, and the state object specified in the call to `SearchAsync`. The `ContactsSearchEventArgs` also has a `Results` property, which is a collection containing contacts that matched the search term.

You're now ready to implement your own `SearchCompleted` event handler. You start by hooking the `SearchCompleted` event from the `Contacts` class. Wire up the `SearchCompleted` event to the `contacts_SearchCompleted` method in the constructor for the `MainPage` class:

```
contacts.SearchCompleted += contacts_SearchCompleted;
```

Now implement the event handler:

```
void contacts_SearchCompleted(object sender, ContactsSearchEventArgs e)
{
   contactsResult.Text =
      string.Format("{0} contacts found", eResults.Count());
   if (results.Count > 1)
      contactsResult.Text += string.Format(", displaying the first match");
   contactControl.Content = e.Results.FirstOrDefault();
}
```

The event args `Results` property is of type `IEnumerable<Contact>`, which allows you to use the Linq extension methods `Count` and `FirstOrDefault`. You use the `Count` extension method to display the number of contacts that matched the search term in the `Text` property of the `contactsResult` `TextBlock`. Next you use the `FirstOrDefault` extension method to set the `Content` property of the `contactControl` `ContentControl`.

The `Contact` class contains more than a dozen different properties and one method. `CompleteName`, `DisplayName`, and `IsPinnedToStart` are the only properties that aren't collections. The remaining properties are collections of `Strings`, `DateTimes`, or more complex objects. Table 4.4 details the complex classes found in the `UserData` namespace that are used by the `Contact` class.

Table 4.4 Contacts related classes

Class	Description
`CompleteName`	The first, middle, and last name of the contact. Nickname, title, and suffix are also provided.
`ContactAddress`	Contains a physical address for the contact in the form of a `CivicAddress` instance. Also contains an `AddressKind` property that specifies whether the address is a home or work address.
`ContactCompanyInformation`	Contains the company name and location, as well as the job title of the contact.
`ContactEmailAddress`	Contains an email address for the contact. Also contains an `EmailAddressKind` property that specifies whether the email address is for a personal or work account.
`ContactPhoneNumber`	Contains a phone number for the contact. Also contains a `PhoneNumberKind` property that specifies whether the phone number is for home, work, a mobile, a fax, or other type of device.

All the classes in this table except `CompleteName` contain an `Accounts` collection specifying the source of the data.

If you run the application now, you won't see any `Contact`-related data. You're setting the `Content` property of the `contactContent` `ContentControl` to an instance of `Contact`. You haven't yet declared any user interface elements to display the `Contact`. The next listing details the XAML markup to display some of the data available for a contact.

Listing 4.10 Displaying contact properties with a DataTemplate and data binding

```
<DataTemplate>
    <StackPanel  Margin="{StaticResource PhoneMargin}">
        <TextBlock Text="{Binding DisplayName,
➥  StringFormat='Display Name: \{0\}'}" />
        <StackPanel Orientation="Horizontal" >
            <TextBlock Text="{Binding CompleteName.FirstName,
```

① **Data binding with StringFormat**

```
➥  StringFormat='Complete Name: \{0\} '}" />
            <TextBlock Text="{Binding CompleteName.MiddleName,
➥  StringFormat='\{0\} '}" />
            <TextBlock Text="{Binding CompleteName.LastName}" />
        </StackPanel>

        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding PhoneNumbers[0].Kind,
➥  StringFormat='\{0\} Number: '}" />
            <TextBlock Text="{Binding PhoneNumbers[0].PhoneNumber}" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="{Binding EmailAddresses[0].Kind,
➥  StringFormat='\{0\} Email: '}" />
            <TextBlock Text="{Binding EmailAddresses[0].EmailAddress}" />
        </StackPanel>

        <TextBlock Text="{Binding Addresses[0].Kind,
➥  StringFormat='\{0\} Address: '}"/>
        <TextBlock Text="{Binding
➥  Addresses[0].PhysicalAddress.AddressLine1}"/>
        <TextBlock Text="{Binding Addresses[0].PhysicalAddress.City}" />

        <TextBlock Text="{Binding Companies[0].CompanyName,
➥  StringFormat='Company: \{0\}'}"/>
        <TextBlock Text="{Binding Companies[0].JobTitle,
➥  StringFormat='Title: \{0\}'}" />
        <TextBlock Text="{Binding Companies[0].OfficeLocation,
➥  StringFormat='Office Location: \{0\}'}" />
    </StackPanel>
</DataTemplate>
```

❷ **First phone number in collection**

The user interface declared in the code listing is not pretty. You've stacked several `TextBlock`s on top of each other, each showing their own bit of data. Each `TextBlock` is bound to one of the properties of the `Contact` object that's referenced by the `contactControls`'s `Content` property. You make use of the `StringFormat` feature available in Silverlight 4's data binding syntax ❶. You choose to display only the first phone number in the `PhoneNumbers` collection by using index syntax in the binding expression ❷, and you use the same index syntax to display email address, street address, and company information.

Run the application now, input a search term, pick a search type, and execute the search by tapping the button. If a matching contact is found, you should see available contact details displayed on the screen. Contact data is only part of the information available through the UserData API. Calendar appointments are also exposed via the UserData APIs.

4.4.2 Reviewing appointments

Calendar records are discovered by executing an asynchronous search against the appointments data store. The UserData API contains the `Appointments` class, which is the façade providing access to each service provider's calendar data. Searches are defined in terms of a date range and can be restricted to a specified `Account`.

TIP Use of the Appointments API requires the `ID_CAP_APPOINTMENTS` capability to be declared in WMAppManifest.xml.

Your sample application allows the user to search for all appointments within a specified date range. The three supported date ranges are all appointments for today, all appointments in the next 7 days, and all appointments in the next 30 days. The user will specify the data range via a series of `RadioButtons` in the second `PivotItem` in your applications. The XAML markup for this is shown in the following listing.

Listing 4.11 XAML markup for the appointments pivot

```
<controls:PivotItem Header="appointments">
    <StackPanel>
        <StackPanel Orientation="Horizontal">
            <StackPanel  Width="350">
                <RadioButton x:Name="todaySearch" Content="Today"
                    IsChecked="True" />
                <RadioButton x:Name="weekSearch" Content="Next 7 Days" />
                <RadioButton x:Name="monthSearch" Content="Next 30 Days" />
            </StackPanel>
            <Button Click="searchAppointments_Click"
                VerticalAlignment="Top">
                <Image Source="/Images/appbar.feature.search.rest.png" />
            </Button>
        </StackPanel>

        <TextBlock Text="Search Result:"
            Style="{StaticResource PhoneTextGroupHeaderStyle}" />
        <TextBlock x:Name="apptsResult" Margin="12" />
        <ContentControl x:Name="appointmentControl">
            <ContentControl.ContentTemplate>
                <DataTemplate>
                </DataTemplate>
            </ContentControl.ContentTemplate>
        </ContentControl>
    </StackPanel>
</controls:PivotItem>
```

① **Relabel item2**

Radio buttons to specify date range ②

③ **Display results**

You place the XAML markup inside the second `PivotItem` and change the header ① from *item2* to *appointments*. Three `RadioButton` controls are added ② to allow the user to specify the date range to use when searching. Next you add the `Button` that the user will use to start a search. The `Button`'s `Click` event is handled by a method named `searchAppointments_Click` which you'll implement in the following pages. Finally you add a few controls to display the results of the search ③. You're leaving the `appointmentControl` `ContentControl` empty for now.

The `Appointments` class provides the `SearchAsync` method and uses the same asynchronous pattern that the `Contacts` class uses. Results are returned via a completed event named `SearchCompleted`. There are four overrides of the `SearchAsync` method. The overrides all accept different combinations of start date, end date, account, and the maximum number of items to be returned. Each of the `SearchAsync` overrides also accepts a state object that calling code can use to pair method calls to completed events.

The `SearchAsync` method limits the number of appointments returned. When you use an override of the `SearchAsync` method that doesn't specify the number of items, the search will be performed using the value defined in the `DefaultMaximumItems` field of the `Appointments` class. The value of `DefaultMaximumItems` is defined to be 100 items.

Before implementing the `searchAppointments_Click` method, you need to declare and construct an instance of the `Appointments` class. Add a field to the `MainPage` class to hold the `Appointments` instance:

```
Appointments appointments = new Appointments();
```

With the appointment instance created you're ready to search the calendar data store. The following listing implements the `searchAppointments_Click` method where you perform the search.

Listing 4.12 Searching for appointments

```
private void searchAppointment_Click(object sender, RoutedEventArgs e)
{
    DateTime start = DateTime.Today;
    DateTime end = DateTime.Today.AddDays(1).AddSeconds(-1);

    if (weekSearch.IsChecked.Value)
        end = end.AddDays(7);
    else if (monthSearch.IsChecked.Value)
        end = end.AddDays(30);

    appointments.SearchAsync(start, end, null);
}
```

❶ **Start search with today's date**

❷ **Adjust end date**

❸ **Execute search**

First you set the `StartDate` using `DateTime.Today`, which will cause the start time to be 12:00 a.m. You set the `EndDate` to be 11:59:59 p.m. by adding one day to `Today`, then subtracting one second ❶. Next you adjust the end date by adding either 7 or 30 days, depending on which `RadioButton` is checked ❷. Finally you call the `SearchAsync` method ❸.

Search results are returned to the application via the `SearchCompleted` event which passes an instance of the `AppointmentsSearchEventArgs` class. The `AppointmentsSearchEventArgs` class exposes `StartTimeInclusive`, `EndTimeInclusive`, and the state object specified in the call to `SearchAsync`. The `AppointmentsSearchEventArgs` also has a `Results` property, which is a collection containing appointments that fall within the specified date range, up to the specified number of items returned.

You're now ready to implement your own `SearchCompleted` event handler. You start by hooking the `SearchCompleted` event from the `Appointments` class. Wire up the `SearchCompleted` event to the `appointments_SearchCompleted` method in the constructor for the `MainPage` class:

```
appointments.SearchCompleted += appointments_SearchCompleted;
```

Now implement the event handler:

```
void appointments_SearchCompleted(object sender,
    AppointmentsSearchEventArgs e)
{
    apptsResult.Text =
        string.Format("{0} appointments found", e.Results.Count());
    if (e.Results.Count() > 1)
        apptsResult.Text += string.Format(", displaying the first match");
    appointmentControl.Content = results.FirstOrDefault();
}
```

You use the Linq extension methods `Count` and `FirstOrDefault` to extract information from the `Results` property, which is of type `IEnumerable<Appointment>`. You set the `Text` property of the `apptsResult` `TextBlock` to a string containing the number of appointments returned by the search. Next you use the `FirstOrDefault` extension method to set the `Content` property of the `appointmentControl` `ContentControl`.

The `Content` property now holds an instance of the `Appointment` class. The `Appointment` class defines several properties providing details about the appointment. Table 4.5 lists each of the `Appointment` properties.

Table 4.5 Appointment properties

Name	Description
`Account`	Details about which service provider is the source of the data.
`Attendees`	A collection of `Attendee` records representing people who were invited to the event. The `Attendee` class has `DisplayName` and `EmailAddress` properties.
`Details`	A `string` value describing the appointment.
`EndTime`	A `DateTime` value.
`IsAllDayEvent`	True if the appointment is flagged as an all-day event.
`IsPrivate`	True if the appointment is flagged as private.
`Location`	A `string` describing the location of the appointment
`Organizer`	The `Attendee` record representing the person who created the appointment
`StartTime`	A `DateTime` value.
`Status`	An `AppointmentStatus` enumeration with the value `Busy`, `Free`, `OutOfOffice`, or `Tentative`
`Subject`	A string describing the subject of the appointment

You'll create a simple user interface that will display a few of the properties from the appointment. The appointment UI will be added to the `appointmentControl` `ContentControl`. Because the `appointmentControl` component contains an `Appointment` instance in its `Content`, you can use data binding to display appointment details with the `ContentControl`'s `DataTemplate`. The following listing implements the appointment user interface.

Listing 4.13 Displaying an appointment

```
<DataTemplate>
  <StackPanel Margin="{StaticResource PhoneMargin}">
    <TextBlock Text="{Binding Subject, StringFormat='Subject: \{0\} '}" />

    <TextBlock Text="{Binding StartTime, StringFormat='Start: \{0\}'}" />
    <TextBlock Text="{Binding EndTime, StringFormat='End: \{0\}'}" />
    <TextBlock Text="{Binding Status, StringFormat='Status: \{0\}'}" />

    <TextBlock Text="{Binding Organizer.DisplayName,
➥ StringFormat='Organizer: \{0\}'}" />

    <TextBlock Text="{Binding Attendees[0].DisplayName,
➥ StringFormat='Attendee: \{0\}'}" />
  </StackPanel>
</DataTemplate>
```

You're displaying the `Appointment` with the same crude technique you used when displaying contact information. You stack several `TextBlocks`, one on top of the other, and then use data binding and string formatting to set their `Text` properties.

4.5 Summary

The phone's security sandbox doesn't allow applications to share data. Sharing data with the built-in native applications can be accomplished with launchers and choosers. Launchers and choosers provide a programmer the means to initiate phone calls, emails, and text messages, and to read and write data to the contacts database. Marketplace tasks provide rich integration with the Windows Phone Marketplace.

A third-party application can also read directly from the contact and appointment database using the UserData APIs. The UserData APIs expose more than just a contact's phone number, email address, and street address. The UserData API opens up the entire database including full name, birthdays, employer data, family members, and web sites.

The UserData APIs also provide access to calendar data—something not possible with launchers and chooser. A third-party application can search for appointments that fall within a given date range or come from a specific service provider.

Carefully consider which of the APIs you use in your applications. The launchers and choosers provide a user experience that exactly matches that of the native applications. Even if your application requires full access to the data exposed by the UserData APIs, consider using the chooser tasks when prompting the user to select a phone number or email address. Once you have the phone number, you can search against the contacts data store to retrieve any other data required by your application.

Remember that executing a launcher or chooser will deactivate, and potentially tombstone, your application. When the user finishes the task and returns to your application, the operating system reactivates your application. Be sure to code defensively and test that your application handles tombstoning and reactivation properly when returning from a launcher or chooser.

You haven't seen the last of launchers and choosers. In chapter six you'll use the `CameraCaptureTask` and `PhotoChooserTask` as we explore Window Phone 7's camera support for media. In chapter 13 you'll use the `BingMapsDirectionsTask`, `BingMapsTask`, and `WebBrowserTask` when you integrate your applications with Internet Explorer and the Bing Maps application.

5

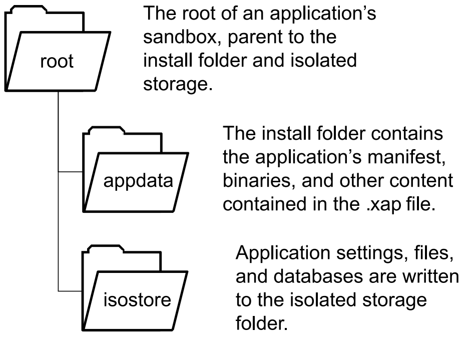
Storing data

This chapter covers

- Working with application settings
- Saving files in isolated storage
- Database operations
- Upgrading databases

Most applications require some form of data storage—from user preferences and user-created data to local caches of data stored in a cloud application or web service. The data storage concepts discussed in this chapter will be familiar to .NET Framework developers, as they're limited versions of the same storage technologies that have existed in the .NET Framework for many years. In this chapter you'll learn about the differences between the storage APIs in the .NET Framework and the storage APIs provided in the Window Phone SDK. For starters, the Windows Phone security model limits storage available to third-party applications to the isolated storage sandbox.

Each application is allotted its own sandbox on the phone, isolated from all other applications and from the operating system. The application's sandbox is separated into two folders, as shown in figure 5.1. The first folder, often referred to as *appdata*, contains the installed application files. The second folder, called *IsolatedStorage* or *isostore*, stores files created or downloaded by the application.



The root of an application's sandbox, parent to the install folder and isolated storage.

The install folder contains the application's manifest, binaries, and other content contained in the .xap file.

Application settings, files, and databases are written to the isolated storage folder.

Figure 5.1 Application file system overview

Developers access isolated storage with either `IsolatedStorageSettings` or `IsolatedStorageFile` from the `System.IO.IsolatedStorage` namespace. Data can also use a relational database in the form of SQL Server CE, accessed via the LINQ to SQL framework found in the `System.Data.Linq` namespace. `IsolatedStorageSettings` provides a simple mechanism for storing data, removing the burden of messing with `FileStreams`. `IsolatedStorageFile` exposes a more robust API for manipulating directories and files. In addition to the IsolatedStorage API, Windows Phone 7.1 includes the LINQ to SQL API for using SQL Server Compact Edition databases.

Windows Phone 7 filesystem restrictions

XNA and .NET (including Compact Framework) developers are used to accessing long-term storage with classes from the `System.IO` namespace including `File`, `FileInfo`, `Directory`, `DirectoryInfo`, `Path`, `FileStream`, and others. Because of the Windows Phone sandbox, many of these classes have been removed, or have been changed to throw `MethodAccessExceptions` when used by applications and games. Silverlight developers should be aware that shared site settings and storage have been removed from `IsolatedStorage` for Windows Phone. Without access to the storage device, or shared site storage files, applications can't share local data with other applications.

The size of isolated storage in Silverlight for the browser applications is controlled by a quota. The quota is initially limited to 1MB. The application can request quota increases, which must be approved by the user. Isolated storage for Windows Phone applications is effectively unlimited. The quota limit defaults to the maximum long value and an exception is thrown if you call the `IncreaseQuotaTo` method.

To demonstrate using the `IsolatedStorage` and LINQ to SQL APIs you're going to build a sample application that reads and writes data using application settings, isolated storage files, and a relational database.

In the next section, you prepare a sample application that will interact with the three different data repositories you create later in the chapter.

5.1 Creating the High Scores sample application

The sample application, shown in figure 5.2, manages a list of high scores for a fictional game. The sample application contains application bar buttons for adding a randomly generated score, as well as a button for deleting the entire high scores table.

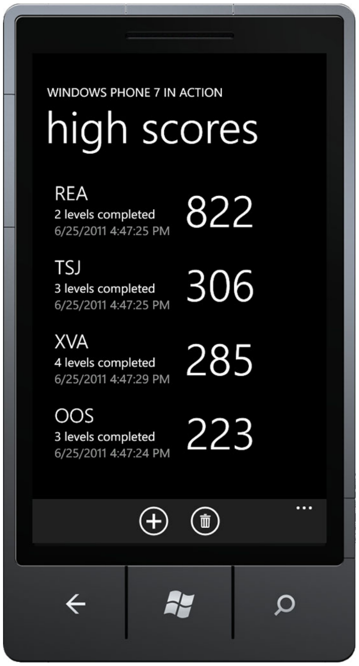
The sample application only shows high scores; you're not going to build a real game in this chapter. If building a game is what you're looking for, jump ahead to chapter 14.

You're going to build the sample application and implement add and delete functionality without worrying about storing the high scores list. You'll then implement reading and writing the high score list using a simple repository interface. You start your sample application by creating a new Windows Phone Application project in Visual Studio. Name the new project `DataStorage`.

The first thing you implement is a class to represent the high score. Create a new class named `HighScore`, and add properties for `Name`, `Score`, `LevelsCompleted`, and `Date`:

```
public class HighScore
{
    public HighScore() { Date = DateTime.Now; }

    public string Name { get; set; }
    public int Score { get; set; }
    public int LevelsCompleted { get; set; }
    public DateTime Date { get; set; }
}
```



WINDOWS PHONE 7 IN ACTION

high scores

REA
2 levels completed
6/25/2011 4:47:25 PM
822

TSJ
3 levels completed
6/25/2011 4:47:25 PM
306

XVA
4 levels completed
6/25/2011 4:47:29 PM
285

OOS
3 levels completed
6/25/2011 4:47:24 PM
223

Figure 5.2 The High Scores sample application

The properties are all simple automatic properties, and the `Date` field is initialized in the class constructor.

5.1.1 Displaying the high score list

High scores are displayed to the user using a `ListBox` on the main page of the application. The following listing shows the `MainPage` XAML markup for the content panel.

Listing 5.1 Displaying high scores

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ListBox x:Name="HighScoresList" Margin="0,0,-12,0">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <Grid Margin="12">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition />
                        <ColumnDefinition />
                    </Grid.ColumnDefinitions>
                    <Grid.RowDefinitions>
                        <RowDefinition />
                        <RowDefinition />
                        <RowDefinition />
                    </Grid.RowDefinitions>
```

① **Divide into 3 rows and 2 columns**

```
                    <TextBlock Text="{Binding Name}"
                        Style="{StaticResource PhoneTextLargeStyle}" />
                    <TextBlock Grid.Row="1"
                        Text="{Binding LevelsCompleted,
➥  StringFormat='\{0\} levels completed'}"
                    Style="{StaticResource PhoneTextNormalStyle}" />
                    <TextBlock Grid.Row="2" Text="{Binding Date}"
                        Style="{StaticResource PhoneTextSubtleStyle}" />

                    <TextBlock Grid.Column="1"
                        Grid.RowSpan="3"
                        VerticalAlignment="Center"
                        Text="{Binding Score}"
                        Style="{StaticResource PhoneTextTitle1Style}" />
                </Grid>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Grid>
```

② **Bind TextBlocks to HighScore properties**

③ **Score spans 3 rows**

The `ListBox` consumes the entire content panel. You define the data template to be a `Grid` divided into three rows and two columns ①. Several `TextBlocks` are arranged inside the `Grid` and are bound to properties of the `HighScore` object ② displayed in the `ListBox`. The `Name`, `Date`, and `LevelsCompleted` values are stacked in the first column and the `Score` value is shown in a larger font, spanning all three rows ③ of the second column.

Next you add two buttons to the application bar for adding new scores and clearing the entire list. Replace the default application bar markup with the XAML shown in the following snippet:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="False">
        <shell:ApplicationBarIconButton Click="add_Click"
            IconUri="/Images/appbar.add.rest.png" Text="add" />
        <shell:ApplicationBarIconButton Click="clear_Click"
            IconUri="/Images/appbar.delete.rest.png" Text="clear" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

As with many of the applications in this book, you're using icons from the Windows Phone SDK, which are installed by the Windows Phone tools to c:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Icons. On 64-bit Windows, the SDK installed in c:\Program Files (x86). Create a project folder named `Images` and add the appbar.add.rest.png and appbar.delete.rest.png files to the folder. For each of the images set the Build Action to Content. More information about the application bar can be found in chapter 10.

The `ApplicationBarIconButton Click` events will be added soon, but first you need to create a collection to hold the high score list in the page's code behind file:

```
ObservableCollection<HighScore> highscores;

public MainPage()
{
    InitializeComponent();
    highscores = new ObservableCollection<HighScore>();
    HighScoresList.ItemsSource = highscores;
}
```

You use an `ObservableCollection` to hold all the high scores. The collection is defined as a field in the `MainPage` class and is instantiated in the constructor. Finally, the collection is assigned to the `ItemSource` property of the `ListBox` you named `HighScoresList`.

5.1.2 Managing the high score list

Now that you have the collection created in code behind and displayed with XAML markup, you need to implement the logic for adding and clearing the list of high scores. A new `HighScore` is created with random values when the user presses the Add Application bar button. The following listing details how a `HighScore` is generated and added to the collection.

Listing 5.2 Adding a `HighScore` with random values

```
Random random = new Random();

private void add_Click(object sender, EventArgs e)
{
    int score = random.Next(100, 1000);
    int level = random.Next(1,5);
    string name = string.Format("{0}{1}{2}", (char)random.Next(65,90),
        (char)random.Next(65,90), (char)random.Next(65,90));

    var highscore = new HighScore { Name = name, Score = score,
        LevelsCompleted = level };

    bool added = false;
    for (int i = 0; i < highscores.Count; i++)
    {
        if (highscores[i].Score < highscore.Score)
        {
            highscores.Insert(i, highscore);
            added = true;
            break;
        }
    }
    if (!added)
        highscores.Add(highscore);
}
```

❶ **Class-level random number generator**

❷ **Generate random values**

❸ **Insert into sorted collection**

You generate new `HighScores` with the help of the `Random` class ❶, a pseudo-random number generator which you place in a new field named `random`. Next you define the `add_Click` event handler. You generate random values for the score, the level, and three characters that make up the name ❷. The random values are used to construct a new `HighScore` object. Finally, you insert the score into the highscores collection using an insertion-sort technique ❸.

Clearing the list is much easier. Inside the `Click` event handler you call the `Clear` method on the collection:

```
private void clear_Click(object sender, EventArgs e)
{
    highscores.Clear();
}
```

The shell of the sample application is now complete. You can run the application, add a few high scores, and clear the list. When you exit the application and restart, the high scores list is empty once again. Since the topic of this chapter is data storage, the sample is far from complete. The application should be storing the high score list and reloading the list when the application restarts.

Throughout the rest of this chapter you'll look at three different methods for storing data—application settings, isolated storage files, and LINQ to SQL. Before we examine the details of each method, you should abstract the data layer from the user interface using a high score repository.

5.1.3 Defining a high score repository

You now have a working application, but the application doesn't store the list of high scores. To hide the details of the data storage implementation, you'll create a high score repository to load, save, and clear the list of high scores. The repository will be defined using an interface. In this section you define the new repository interface and modify the sample application to use the interface methods to access the high score data.

Add a new `Interface` to the project using Visual Studio's Add New Item feature. Name the interface `IHighScoreRepository`. The new interface declares `Load`, `Save`, and `Clear` methods:

```
interface IHighScoreRepository
{
    List<HighScore> Load();
    void Save(List<HighScore> highScores);
    void Clear();
}
```

You now update the `MainPage` class to use the repository. `MainPage` should initialize the high score collection from the repository during construction. `MainPage` should also call the `Save` method when new `HighScores` are generated and the `Clear` method when the collection is cleared. Add a repository field to `MainPage` class:

```
IHighScoreRepository repository;
```

You construct the repository in the constructor and load the collection from the repository:

```
public MainPage()
{
    InitializeComponent();
    repository = new HighScoreSettingsRepository();
    highscores = new ObservableCollection<HighScore>(repository.Load());
    HighScoresList.ItemsSource = highscores;
}
```

In this code snippet, you construct a `HighScoreSettingsRepository` which is the first of the concrete implementations of `IHighScoreRepository` that you'll build later in the chapter.

The best time to save the list of high scores is after adding a new `HighScore` to the collection. Add the following line of code to the end of the `add_Click` method:

```
repository.Save(highscores.ToList());
```

The repository's `Clear` method should be called when the `ObservableCollection` is cleared. Add a call to the `Clear` method to the end of the `clear_Click` method:

```
repository.Clear();
```

With the call to the `Clear` method, all of the repository functionality has been wired up. The application requires a concrete repository, and your first one will use application settings to store the high score data.

5.2 Storing data with application settings

Application settings provide a convenient API to store user preferences, application state, or other forms of data. Application settings can store both simple and complex objects using key-value pairs. Application settings are accessed through a static property on the `IsolatedStorageSettings` class. The property, named `ApplicationSettings`, returns an instance of the `IsolatedStorageSettings` class. `IsolatedStorageSettings` implements a `Dictionary` interface with each setting paired with a unique key.

Objects placed in the settings dictionary must be serializable. When objects aren't serializable, an exception will be thrown when the settings are written to the storage device. Application settings are stored in a file written to isolated storage, as shown in figure 5.3.

NOTE Any action that clears isolated storage will remove the file, and application settings will be lost. These actions include uninstalling an application and using the Rebuild Solution command in Visual Studio.

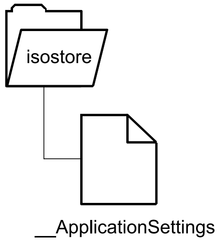


Figure 5.3 Application settings are written to a file in the isolated storage folder.

`IsolatedStorageSettings` can be found in the `System.IO.IsolatedStorage` namespace, and you'll need to add a `using` statement to any code file that makes use of application settings.

Your first concrete implementation of the `IHighScoreRepository` will use application settings to store the list of `HighScore` objects. Create a new class named `HighScoreSettingsRepository` that implements the repository interface:

```
public class HighScoreSettingsRepository : IHighScoreRepository
```

You start by implementing the `Load` method for the new repository. Within the `Load` method, you declare a list of `HighScore` objects and call the `IsolatedStorageSettings` method `TryGetValue` to initialize the list:

```
public List<HighScore> Load()
{
    List<HighScore> storedData;
    if (!IsolatedStorageSettings.ApplicationSettings.
        TryGetValue("HighScores", out storedData))
    {
        storedData = new List<HighScore>();
    }
    return storedData;
}
```

If a key-value pair doesn't exist in application settings, `TryGetValue` will return false, and you create an empty list so that you always return a valid collection.

Next you implement the `Save` method. You add the list of high scores to application settings using the `Item` property of the dictionary:

```
public void Save(List<HighScore> highScores)
{
    IsolatedStorageSettings.ApplicationSettings["HighScores"] = highScores;
    IsolatedStorageSettings.ApplicationSettings.Save();
}
```

Values placed in the application settings dictionary aren't immediately written to disk. The values are usually saved when the application hosting the settings is terminated. Writes can be forced using the `Save` method.

Finally you implement the `Clear` method by making a call to the settings class's `Remove` method. The `Remove` method will clear out just the `HighScores` key, but will leave all other application settings data intact. You also force a write of the data by calling the `Save` method:

```
public void Clear()
{
    IsolatedStorageSettings.ApplicationSettings.Remove("HighScores");
    IsolatedStorageSettings.ApplicationSettings.Save();
}
```

The `IsolatedStorageSettings` class also implements a `Clear` method. The `Clear` method will remove every key-value pair stored in the settings dictionary. You could've used the `Clear` method since you only have a single key-value pair. Using the `Clear` method wouldn't be appropriate in any applications using multiple key-value pairs.

Your repository implementation is complete and you can now use the settings repository in your application. Run the application now, add a few high scores to the list, and exit with the Back button. Restart the application and this time you should see the high scores reloaded and displayed in the user interface. Saving data to application settings is simple and only requires a serializable object to be stored in the settings dictionary. When your application requires a more complex data model you should consider writing the data to files in isolated storage.

5.3 Serializing data to isolated storage files

Access to the filesystem is restricted on the Windows Phone. Instead of accessing the file system with the `Directory` and `File` class found in the `System.IO` namespace, Window Phone developers use `IsolatedStorageFile` found in the `System.IO.IsolatedStorage` namespace. `IsolatedStorageFile` is the file system for an application and provides a basic file system API for managing files and directories. Windows Phone applications aren't allowed to use the traditional `System.IO` classes for managing files and directories, nor are they allowed to see the filesystem outside the sandbox. The file system methods provided by `IsolatedStorageFile` are described in table 5.1. Only one instance of `IsolatedStorageFile` exists on the Windows Phone and is accessible via the `GetUserStoreForApplication` static method.

Table 5.1 Filesystem management methods provided by IsolatedStorageFile

Method	Description
`CreateDirectory`	Create a new directory in the isolated file store.
`CreateFile`	Create a new empty file in the isolated file store, and return a `FileStream`.
`DeleteDirectory`	Delete a directory in the isolated file store. The directory must be empty.
`DeleteFile`	Delete a file in the isolated file store.
`DirectoryExists`	Determine whether a directory exists.
`FileExists`	Determine whether a file exits.
`GetDirectoryNames`	Returns the names of the subdirectories in a specified directory.
`GetFileNames`	Returns the names of the files in a specified directory.
`OpenFile`	Open or create a file and return a `FileStream`.
`Remove`	Delete every file and directory in the isolated file store.

You're going to create a high score repository that reads and writes data to an XML file using the `IsolatedStorageFile` API. As shown in figure 5.4, the data file will be named HighScores.xml, and stored in a new folder named HighScores.

Add a new class to the sample application project, naming the class `HighScoreFileRepository`. The new class should implement the `IHighScoreRepository` interface:

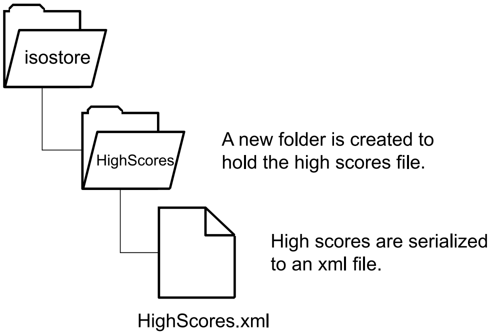

isostore
HighScores
A new folder is created to hold the high scores file.
High scores are serialized to an xml file.
HighScores.xml

Figure 5.4 High scores are written to an XML file located in a new folder in isolated storage.

```
public class HighScoreFileRepository : IHighScoreRepository
```

In the application settings repository you built in the last section, the list of high scores was implicitly serialized by the framework. In the file storage repository, you're responsible for serializing the list. To help with serialization, you're going to use the `XmlSerializer`.

5.3.1 Serializing high scores with the XmlSerializer

The `XmlSerializer` class is found in the `System.Xml.Serialization` namespace. The assembly containing the `XmlSerializer` class isn't automatically added to the project references by the Windows Phone Application project template. To use `XmlSerializer` you must manually add an assembly reference to `System.Xml.Serialization.dll`.

You'll use `XmlSerializer` in both the `Load` and `Save` methods of the new `HighScoreFileRepository` class. The implementation of the `Load` method is shown in the next listing.

Listing 5.3 Loading scores from a file

```
using System.Xml.Serialization

public List<HighScore> Load()
{
   List<HighScore> storedData;
   using (IsolatedStorageFile storage =
      IsolatedStorageFile.GetUserStoreForApplication())
   {
      if (storage.DirectoryExists("HighScores") &&
         storage.FileExists(@"HighScores\highscores.xml"))
      {
         using (IsolatedStorageFileStream stream =
            storage.OpenFile(@"HighScores\highscores.xml", FileMode.Open))
         {
            XmlSerializer serializer =
               new XmlSerializer(typeof(List<HighScore>));
            storedData = (List<HighScore>)serializer.Deserialize(stream);
         }
      }
      else
      {
         storedData = new List<HighScore>();
      }
   }
   return storedData;
}
```

① **Does file exist?**

② **Deserialize list**

③ **Create empty list**

The first operation obtains a reference to the isolated storage associated to the application. The first time the application runs, isolated storage will be empty so you need to check whether the file highscores.xml exists in isolated storage in a directory named HighScores ①. If the file does exist, it's opened and data is read from a file stream ②. An `XmlSerializer` class is created and the `Deserialize` method is used to read the persisted list and assign it to the return variable. If neither the directory nor the file exists, you create a new empty list ③.

What if the HighScores directory doesn't exist? You know that it won't exist the first time you run the application. The directory must be created by your code. The file repository's `Save` method, shown in the next listing, will create the directory if it doesn't exist.

Listing 5.4 Saving the high scores to a file

```
public void Save(List<HighScore> highscores)
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (!storage.DirectoryExists("HighScores"))
        {
            storage.CreateDirectory("HighScores");
        }
        using (IsolatedStorageFileStream stream =
            storage.CreateFile(@"HighScores\highscores.xml"))
        {
            XmlSerializer serializer =
                new XmlSerializer(typeof(List<HighScore>));
            serializer.Serialize(stream, highscores);
        }
    }
}
```

① **Create directory**

② **Create or overwrite file**

③ **Serialize list**

If the directory doesn't exist, such as the first time the application runs, the `Save` method adds a new directory to the root of isolated storage. The code first uses `DirectoryExists` and if a `false` value is returned, calls `CreateDirectory` ①. Next the highscores.xml file is created. `IsolatedStorageFile.CreateFile` ② creates a new file, or overwrites an existing file, and returns an opened `IsolatedStorageFileStream`. The file stream will be used to write data to the file, and should be closed when it's no longer needed. You place the code using the stream inside a `using` block, which will automatically call `close` and clean up the file handle for you. Finally, the list of high scores is written to the XML file using the `Serialize` method of the `XmlSerializer` class ③.

5.3.2 Deleting files and folders

The last repository method that you need to implement is `Clear`. The file repository implements `Clear` by deleting the XML file and the `HighScores` folder, as seen in listing 5.5. Once created, the HighScores.xml file and the HighScores folder will continue to exist in isolated storage. Files and folders in isolated storage are deleted when a user uninstalls an application, but they remain unchanged when a user upgrades an application to a new version.

Listing 5.5 Clearing the high score data

```
public void Clear()
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (storage.FileExists(@"HighScores\highscores.xml"))
            storage.DeleteFile(@"HighScores\highscores.xml");

        if (storage.DirectoryExists("HighScores"))
            storage.DeleteDirectory("HighScores");
    }
}
```

➊ **Delete XML file**

➋ **Delete directory**

You've already seen how to use the `FileExists` and `DirectoryExists` methods. If either the XML file or the HighScores directory exist, you delete them. Files are deleted with the `DeleteFile` ➊, which requires a full path. Directories are deleted with the `DeleteDirectory` ➋ method. The full path is also required by `DeleteDirectory`.

An alternative implementation of `Clear` could be written using `IsolatedStorageFile`'s `Remove` method. `Remove` is a static method and deletes all files and folders in an application's file store:

```
public static void Clear()
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        storage.Remove();
    }
}
```

Extra care should be employed when using the `Remove` method. If your application stores any other data, including application settings, that data will be lost when `Remove` is called.

The final step before running the application is to update the `MainPage` constructor to instantiate a `HighScoreFileRepository`:

```
repository = new HighScoreFileRepository();
```

The file repository demonstrates file and directory management within the isolated file store available to an application. Some data models match nicely with a file-based storage solution. Other data models work better with a relational database solution.

5.4 Working with a database

Windows Phone 7.1 SDK ships with a built-in relational database engine allowing applications to store data in a local database. The built-in database engine is a version of SQL Server Compact Edition. The bad news is that you can't use raw SQL to interact with the database. The data access API for working with local databases is LINQ to SQL.

> **NOTE** LINQ to SQL for Windows Phone only works with local databases. If your application needs to work with a remote database, you should consider using the OData API. You can learn more about OData at www.odata.org.

LINQ to SQL is an *object relational mapping (ORM)* technology first introduced with version 3.5 of the .NET Framework for desktop and server applications. Applications work with objects defined in a programming language, whereas relational databases work with tables, records, and fields defined by the database schema. An ORM is responsible for mapping the objects in the application with rows and fields in a database table. In your sample application, you work with `HighScore` objects. LINQ to SQL will transform `HighScore` objects into rows in the `HighScores` table of your database, and rows in your database to `HighScore` objects.

The LINQ to SQL libraries are found in the assembly named `System.Data.Linq.dll`. When working with LINQ to SQL you'll need to add a reference to the assembly. The `System.Data.Linq` assembly contains the following namespaces:

- `System.Data.Linq`
- `System.Data.Linq.Mapping`
- `Microsoft.Phone.Data.Linq`
- `Microsoft.Phone.Data.Linq.Mapping`

The `System.Data.Linq` and `System.Data.Linq.Mapping` namespaces contain a subset of the LINQ to SQL APIs that are found in the desktop libraries. The `Microsoft.Phone.Data.Linq` and `Microsoft.Phone.Data.Linq.Mapping` namespaces contain phone-specific extensions to LINQ to SQL.

Throughout the remainder of this chapter you'll learn how to modify your sample application to use a local database. You'll learn how to read and write high score data to the local database using LINQ expressions. You'll also learn how to create and delete databases, and work with read-only databases that are included in the application's XAP deployment package. First you need to learn how to define your database schema using LINQ to SQL classes and attributes.

5.4.1 Attributing your domain model

The Windows Phone implementation of LINQ to SQL allows two different methods of defining a local database schema. The first method, which we cover in this section, uses attributes attached to the classes defined in your code. The second method, which is beyond the scope of this book, uses XML files to define the mapping.

LINQ to SQL provides attribute classes to define tables, columns, and indexes. One-to-one, one-to-many, and many-to-many relationships between tables are also declared using attributes. The LINQ to SQL attributes that are supported by Windows Phone are listed in table 5.2.

Table 5.2 LINQ to SQL attributes

Attribute	Description
`Association`	Declares an association between two classes, resulting in a relationship between two tables in the database.
`Column`	Declares a column in a table and maps the fields of an object to the column. The `Column` attribute has several properties that are used to describe the column.

Table 5.2 LINQ to SQL attributes *(continued)*

Attribute	Description
`Index`	Declares an index on a table. Multiple indexes can be declared for the same table. The `Index` attribute is found in the `Microsoft.Phone.Data.Linq.Mapping` namespace.
`Table`	Declares a table in the database and identifies the type object stored in the table. The database table name can be customized with the attribute's `Name` property.

With the `Table`, `Column`, `Index`, and `Association` attributes, an entire database schema can be created. A multi-column index can be created by providing a comma-separated list of names for the `Columns` property.

The sample high score application is fairly simple, and can be implemented using a single database table. You start the implementation by adding a few attributes to the existing `HighScore` class. The new `HighScore` code is shown in the following listing.

Listing 5.6 Updating the `HighScore` class

```
using System.Data.Linq.Mapping;
using Microsoft.Phone.Data.Linq.Mapping;

[Table]
[Index(Columns="Score")]
public class HighScore
{
    public HighScore() { Date = DateTime.Now; }

    [Column] public string Name { get; set; }
    [Column] public int Score { get; set; }
    [Column] public int LevelsCompleted { get; set; }
    [Column] public DateTime Date { get; set; }

    [Column(IsPrimaryKey = true, IsDbGenerated = true)]
    public int Id { get; set; }
}
```

① **Use mapping namespaces**

② **Define index**

③ **Create primary key**

First you add `using` statements for the namespaces ① `System.Data.Linq.Mapping` and `Microsoft.Phone.Linq.Mapping` so you can easily use the `Table`, `Index`, and `Column` attributes. You start defining the database schema by adding the `Table` attribute to the `HighScore` class. You also add the `Index` attribute ② to the `HighScore` class, declaring a database index using the `Score` column. Next you add a `Column` attribute to each of the fields in the `HighScore` class. Finally you create a new `Id` field ③ declaring that the field is the primary key for the table and that its value is created by the database.

The attributed `HighScore` class defines how the HighScore table will be built in the database. Each row in the table will be mapped to a `HighScore` object by LINQ to SQL. You might be wondering how LINQ to SQL combines all tables together to define the database, and how you access the tables from code. The database itself is represented by LINQ to SQL with the data context.

5.4.2 Defining the data context

Each LINQ to SQL database is represented by a custom data context implementation. The data context is the API used to access data in the database. The data context defines the tables, caches reads, and tracks changes to objects. The data context knows what has changed, and performs the appropriate SQL update statements. Your code tells the data context when to add a new object or delete an existing object, and the data context issues the appropriate insert or delete commands. Changes, insertions, and deletions are queued up in memory until LINQ to SQL is asked to submit the changes to the database, which happens when the `SubmitChanges` method is called. The data context performs all changes using transactions.

You'll look at how to use the data context in create, read, update, and delete (CRUD) operations later in the chapter. First let's look at how to define a data context for your HighScores database. Each custom data context is derived from the `DataContext` base class found in the `System.Data.Linq` namespace. In this sample application, you create a new class named `HighScoresDataContext` and derive it from the `DataContext` class:

```
using System.Data.Linq;

public class HighScoresDataContext : DataContext
{
    public Table<HighScore> HighScores;
    public HighScoresDataContext(string path) : base(path) { }
}
```

For each table that should be created in the database, there must be a matching field in the `DataContext` class. The field will be of type `Table<T>`, where `T` is the LINQ to SQL attributed class that will be stored in the table. In the sample application, you only have one table to hold `HighScore` objects, so your data context defines a `Table<HighScore>` field.

The only other change you make to the `HighScoresDataContext` class is to implement a constructor. The constructor will accept a connection string and pass it along to the base class. The connection string can provide database details such as the filename of the database, the maximum size of the database, the read/write mode used to open the database file, and other options. Sample connection strings are shown in table 5.3.

The simplest connection string requires only a file name. The `DataContext` class will look in the root folder of the application's isolated storage for a file with the given name. Database files can exist in folders other than the root folder. Read-only database files can also be read from the application's installation folder using the appdata: prefix. The default maximum value for a database is 32 megabytes.

Now that you have a data context class and have defined database mapping attributes on the HighScore class, you're ready to create the database.

Table 5.3 Sample database connection strings

Connection string	Description
/file1.sdf	Open the database located in the isolated storage file named file1.sdf. Open the database in read write mode.
isostore:/file1.sdf	Same as /file1.sdf.
/folder1/file1.sdf	Open the database located in isolated storage in the folder named folder1, in the file named file1.sdf.
datasource='appdata:/file1.sdf';mode=read only	Open the database located in the application's install folder, in the file named file1.sdf. Open the database in read-only mode.
datasource='/file1.sdf'; max database size=512	Open the database located in the isolated storage file named file1.sdf. Allow the database to grow to 512 megabytes.

For a complete list of connection string parameters, see the LINQ to SQL for Windows Phone documentation on MSDN.

5.4.3 Creating the database

You've defined your database schema and data context, and are ready to use the database in your HighScores sample application. You'll be loading and saving high scores using the same `IHighScoreRepository` interface you've been using throughout this chapter. You're going to add a new repository class to your project. Name the new class `HighScoreDatabaseRepository` and implement the repository interface.

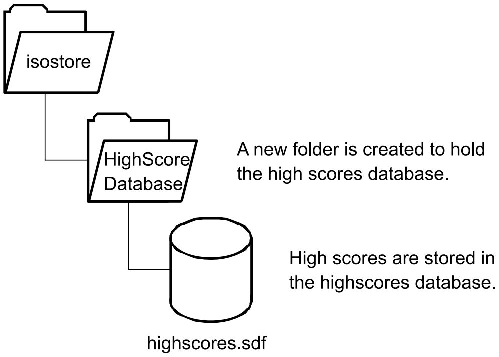

isostore
HighScore Database
A new folder is created to hold the high scores database.
High scores are stored in the highscores database.
highscores.sdf

Figure 5.5 Highscores are written to a database located in a new folder in isolated storage.

Your new repository class will use a `HighScoresDataContext` to read and write high scores to a database file in isolated storage. As shown in figure 5.5, the database file will be named highscores.sdf, and will be placed in a folder named HighScoreDatabase.

Before you create the database, you declare the data context as a field in the repository, and instantiate it in the repository constructor. The following listing details the repository class definition and constructor method.

Listing 5.7 Creating the high scores database

```
public class HighScoreDatabaseRepository : IHighScoreRepository
{
    HighScoresDataContext db;
    public HighScoreDatabaseRepository()
    {
        using (IsolatedStorageFile storage =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            if (!storage.DirectoryExists("HighScoreDatabase"))
            {
                storage.CreateDirectory("HighScoreDatabase");
            }
        }

        db = new HighScoresDataContext(
            @"isostore:/HighScoreDatabase/highscores.sdf");

        if (!db.DatabaseExists())
        {
            db.CreateDatabase();
        }
    }
}
```

❶ **Create folder in isolated storage**

❷ **Instantiate data context**

❸ **Create database file**

Before you create a new data context, you check whether the HighScoreDatabase folder exists, and if it doesn't, you create the folder ❶. Next you instantiate a new `HighScoresDataContext` object, specifying a file named highscores.sdf in the HighScoreDatabase folder ❷. Finally you check whether the file has been created using the `DatabaseExists` method. If the database doesn't yet exist, you ask the data context to create one ❸.

You're now ready to implement the remaining repository methods. The `Load`, `Save`, and `Clear` methods will use database create, read, update, and delete (CRUD) operations provided by the data context.

5.4.4 CRUD operations

Usually, when working with databases, the developer has to keep track of which items are new, which items have been updated, and which items have been deleted. LINQ to SQL frees the developer from these tedious bookkeeping tasks. The data context tracks which objects have been updated, and will issue the appropriate update statements when changes are submitted. New objects are created in the normal fashion (using the new keyword) and are inserted into the table. To delete an object, the developer simply removes the object from the table.

READING DATA

We're getting ahead of ourselves—before you can work with objects loaded from a database, you must first be able to read them from the database. Your `HighScoreDatabaseRepository` class reads high scores from the database when the application calls the `Load` method. The LINQ to SQL table is a queryable collection, which allows you to use LINQ expressions to retrieve data from the table:

```
public List<HighScore> Load()
{
    var highscores = from score in db.HighScores
```

```
                      orderby score.Score descending
                      select score;
    return highscores.ToList();
}
```

In the `Load` method, you query the `HighScores` table using a LINQ expression built with the integrated query language. You select `HighScore` objects and sort them in a descending order of score. You then convert the collection to a list and return.

Behind the scenes, the LINQ to SQL framework converted the LINQ expression into a true SQL select statement. The SQL statement was sent to the SQL Server CE database, which returned a result set. LINQ to SQL then transformed the result set into a collection of `HighScore` objects. The list of high scores is returned to the application and displayed in the user interface just like when you used application settings and isolated storage files.

CREATING NEW HIGH SCORES

The user of the application creates a new high score by tapping the Add button. The button's click handler creates a new `HighScore` object and inserts it into the list of high scores, then asks the repository to save the list. LINQ to SQL tracks high score objects that it creates, but it doesn't automatically track objects created by application code. A new object can be added to the list of tracked objects using either the `InsertOnSubmit` or `InsertAllOnSubmit` methods provided by the `Table` class.

The `InsertOnSubmit` method accepts a single object. The new object will be added to the internal list of tracked objects, and will be added to the database the next time the `SubmitChanges` method is called. `InsertAllOnSubmit` works with a collection of new objects.

The `IHighScoreRepository` interface doesn't provide a mechanism for registering new objects with the repository and you don't need to change the interface. Instead, you can add logic to the `Save` method to detect when unmanaged objects have been added to the collection:

```
public void Save(List<HighScore> highScores)
{
    var newscores = highScores.Where(item => item.Id == 0);
    db.HighScores.InsertAllOnSubmit(newscores);
    db.SubmitChanges();
}
```

You detect when a `HighScore` object is new by looking at the value of the `Id` field. You added the `Id` field to the `HighScore` class because the database required a primary key, and you declared that the field's value would be generated by the database. This means that objects with a field value of zero have never been added to the database. You use a regular LINQ `Where` expression to find new `HighScore` objects in the list. Once you've identified new objects, you use the `InsertAllOnSubmit` method to register them as new objects with the LINQ to SQL table.

You wrap up your implementation of the `Save` method by calling `SubmitChanges` on the `DataContext`. The `DataContext` will generate SQL insert statements for any new records. Once the new scores are submitted to the database, LINQ to SQL will read the generated Id values and update the `HighScore` objects so that their `Id` fields won't be zero.

Note that if you'd changed a value in an existing field, the changes would also have been submitted to the database with a SQL update statement. We'll take a close look at how LINQ to SQL updates changed data in a couple of pages. First we look at how to delete data.

DELETING DATA

The sample application allows the user to reset the list of high scores by tapping the Clear button. The button's click handler calls the repository's `Clear` method. LINQ to SQL tracks high score objects that it reads from the database, but it doesn't automatically know when to remove an object from the database. An object can be removed from the database using either the `DeleteOnSubmit` or `DeleteAllOnSubmit` methods provided by the `Table` class.

When deleting a single object, use the `DeleteOnSubmit` method. The deleted object will be added to the internal list of tracked objects, and will be removed from the database the next time the `SubmitChanges` method is called. `DeleteAllOnSubmit` works with a collection of objects.

The `IHighScoreRepository` interface assumes that `Clear` will delete all high score records. Your database implementation of the `Clear` method reads all the high scores from the `HighScores` table, and then calls `DeleteAllOnSubmit` to delete every high score:

```
public void Clear()
{
    var scores = from score in db.HighScores
                 select score;

    db.HighScores.DeleteAllOnSubmit(scores);
    db.SubmitChanges();
}
```

When the `SubmitChanges` method is called, the appropriate SQL delete statements are generated and sent to the database.

The `HighScoreDatabaseRepository` is now fully implemented. Change the `MainPage` constructor to use the new repository:

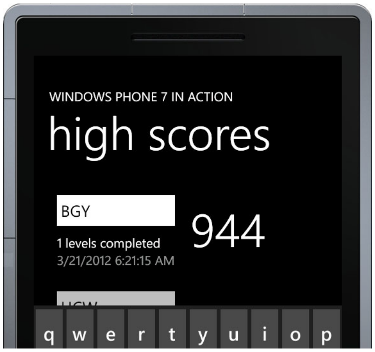
```
repository = new HighScoreDatabaseRepository();
```

Run the application and add a few new high scores. Exit and restart the application and you should see the high scores reloaded from the database and displayed in the user interface. Clear the list and restart the application, and you should see a blank screen.

Now that the basic create, read, and delete operations are working, let's take a closer look at how LINQ to SQL implements update operations.

UPDATING DATA

By default, the database context stores two instances of each object. The first object is returned as part of the query and the second object is an unchanged internal copy. It uses the internal copy to determine when an object has been changed. When the `Submit` method is called, LINQ to SQL walks through each object it knows about and compares it to the internal copy. If the two objects differ in any way, LINQ to SQL generates a SQL update expression to store the changes.


WINDOWS PHONE 7 IN ACTION

high scores

BGY

944

1 levels completed

3/21/2012 6:21:15 AM

Figure 5.6 Editing the name associated with a high score

> **NOTE** If you're not making changes to a database, you can improve performance by setting the `DataContext`'s `ObjectTrackingEnabled` property to false. When object tracking is false, the `DataContext` doesn't generate internal copies of tracked objects.

In order to demonstrate how to update objects with LINQ to SQL, you're going to add an editing feature to your sample application. You're going to allow the user to edit the name associated with the `HighScore` object, as shown in figure 5.6.

The user will edit the name with a standard `TextBox` control. In MainPage.xaml, replace the `TextBlock` that displays the name with a `TextBox`:

```
<TextBox Text="{Binding Name, Mode=TwoWay}" />
```

Now add an application bar button that the user can tap to force a save of changes:

```
<shell:ApplicationBarIconButton Click="save_Click"
    IconUri="/Images/appbar.save.rest.png" Text="save" />
```

Finally, implement the `save_Click` event handler. Inside the event handler, you call the repository's `Save` method just like you do when a new high score is generated:

```
private void save_Click(object sender, EventArgs e)
{
    var nameInput = FocusManager.GetFocusedElement() as TextBox;
    if (nameInput != null)
        nameInput.GetBindingExpression(TextBox.TextProperty)
            .UpdateSource();
    repository.Save(highscores.ToList());
}
```

You might wonder what the first few lines of the snippet do. The `TextBox` control has an interesting quirk when updating bound data. The `TextBox` control tries to be efficient with data binding updates and may not have updated the high score's `Name` field when the `save_Click` event handler is called. In this snippet you get the `TextBox` that has focus and ask for the `BindingExpression` object connected to the control's `Text` property. You use the Binding Expression's `UpdateSource` method to ensure the name is copied from the `TextBox` to the `HighScore` object before you call the `Save` method.

You don't need to change the repository's `Save` method implementation. LINQ to SQL is maintaining an internal copy of the change `HighScore` object. When the `SubmitChanges` method is called, LINQ to SQL figures out which high scores have been updated, and saves the change objects to the database.

Developers can help the database context use fewer resources by using objects that implement the `INotifyPropertyChanged` interface. When serving up objects that report their own changes, LINQ to SQL doesn't generate an internal copy of the object. Instead it adds an event handler that listens to the `PropertyChanged` event. When the property changed event is raised, LINQ to SQL records the change internally. Then when the `SubmitChanges` method is called, the `DataContext` uses the list of changed objects to generate the appropriate SQL update expressions.

LINQ to SQL leverages the underlying LINQ framework to provide powerful capabilities when working with collections and databases. Though describing the full features of LINQ is beyond the scope of this book, there are a couple more LINQ features you should understand. LINQ expressions are used to search for records in the database. We're going to look at a couple simple alternative query expressions before discussing how to use pre-compiled query expressions to improve application performance.

5.4.5 Searching for data

Your HighScore sample application uses a single query expression to return an ordered list of every high score in the database. What if you wanted a list of the scores for the third level in your fictional game? If you were writing a traditional SQL expression, you'd add a `WHERE` clause comparing the `LevelsCompleted` column with the value 3. LINQ to SQL allows you to use a similar technique:

```
var highscores = from score in db.HighScores
                 orderby score.Score descending
                 where score.LevelsCompleted == 3
                 select score;
```

In this snippet, you've added a where clause to the LINQ expression. Note that this where clause uses C# comparison operators. You're going to add support for this scenario to the `IHighScoreRepository` interface. Open the file containing `IHighScoreRepository` and alter the `Load` method declaration to accept a level number:

```
List<HighScore> Load(int level = 0);
```

Here you're using C# optional argument syntax to specify a default value for the level argument. You set the default level value to zero, which means return all levels. When the level isn't zero, you should return only the high scores for the requested level. You accomplish this by using a where clause in the LINQ expression. Update the database repository's `Load` method as shown in the next listing.

Listing 5.8 Loading scores for specific levels

```
public List<HighScore> Load(int level = 0)
{
   IEnumerable<HighScore> highscores;
```

❶ **Update arguments list**

```
    if (level == 0)
    {
        highscores = from score in db.HighScores
                     orderby score.Score descending
                     select score;
    }
    else
    {
        highscores = from score in db.HighScores
                     orderby score.Score descending
                     where score.LevelsCompleted == level
                     select score;
    }
    return highscores.ToList();
}
```

❷ **Default query**

❸ **Query including specified level**

You begin updating the Load method in the `HighScoreDatabaseRepository` by adding the `level` argument ❶. Next you wrap the existing query ❷ with an `if` statement, executing the default query when the requested level has the value of zero. Finally you execute a query containing a where expression ❸ when a specific level is requested.

Since your application always uses the same two queries, you can improve performance a bit by compiling the queries once, and using compiled queries in your load method.

5.4.6 Compiling queries

When LINQ to SQL encounters a query expression, the framework parses the expression, turning LINQ syntax into a SQL statement. This parsing must be performed every time the query is executed. To minimize the performance hit of constantly reparsing the same expression, LINQ provides a mechanism to compile an expression once, and reuse it over and over.

Compiled LINQ expressions are represented in code with the `Func<TResult>` class. The `Func` class is a form of `Delegate`, representing a reference to a method that returns the type specified with the `TResult` generic type parameter. There are several variations of the `Func` class, allowing for anywhere from zero to four parameters.

You need two compiled queries, which means you need two member `Func` fields to represent them:

```
Func<HighScoresDataContext, IOrderedQueryable<HighScore>> allQuery;
Func<HighScoresDataContext, int, IQueryable<HighScore>> levelQuery;
```

The first compiled query represents the search for all high scores and is a `Func` class that accepts one argument, a `HighScoresDataContext`, and returns an `IOrderedQueryable`, which is an ordered collection `HighScore` objects. The second compiled query represents the search for the high scores for a specified level and is a `Func` class that accepts two arguments—the data context and the specified level.

The queries are compiled using the `CompiledQuery` utility class found in the `System.Data.Linq` namespace. This class is part of the LINQ to SQL implementation and provides a `Compile` method. You compile the two queries in the `HighScoreDatabaseRepository` constructor. The `allQuery` `Func` is generated with the following code snippet:

```
allQuery = CompiledQuery.Compile(
    (HighScoresDataContext context) =>
        from score in context.HighScores
        orderby score.Score descending
        select score);
```

The second compiled query is generated with a similar expression, but includes a `level` argument and a `where` clause in the query:

```
levelQuery = CompiledQuery.Compile(
    (HighScoresDataContext context, int level) =>
        from score in context.HighScores
        orderby score.Score descending
        where score.LevelsCompleted == level
        select score);
```

With the newly compiled queries at your disposal, you can simplify the `Load` method. The new `Load` method implementation uses the `allQuery` and `levelQuery` Func objects instead of inline LINQ expressions:

```
if (level == 0)
{
    highscores = allQuery(db);
}
else
{
    highscores = levelQuery(db, level);
}
```

If you think using a compiled query looks like calling a method, you're right. Compiled queries are instances of a `Func`, and the `Func` class is a form of a delegate. Delegates are references to methods.

5.4.7 Upgrading

When building applications and databases, it's rare that the data model is perfectly designed before the first release. Applications evolve, either because of new features that weren't thought of during initial design, or because existing features didn't quite solve the problems the application was built to solve. What do you do when you need to change the database schema for an existing database when the user installs a new version of your application?

Database upgrade algorithms usually involve a series of SQL statements that alter and drop tables and columns in the database. The database support in Windows Phone 7 doesn't allow SQL statements to be issued directly against the database. The Windows Phone LINQ to SQL implementation supports a few database upgrade scenarios. These scenarios are focused on additions to the database schema. New tables can be added to the database. New columns and indexes can be added to existing tables. New associations between tables can also be added to the database schema.

To demonstrate updating an existing database, let’s pretend that version 1.0 of your application has already been released and users have installed the application on their phones. In version 1.1 of the application, you’re going to add a field to the `HighScore` class, which corresponds to adding a column to the database used to store the sample application data. When the user upgrades from version 1.0 to 1.1 of your application, the data stored in the application’s IsolatedStorage folder is left intact. The first time version 1.1 of the application runs, it’ll execute code to upgrade the database created by version 1.0 of the application.

To see the upgrade code in action, make sure you run the application and create the database before making the modifications described in this section. Once you have an existing database, open up the `HighScore` class and add a new `Difficulty` field:

```
[Column] public string Difficulty { get; set; }
```

Database updates are performed using the `DatabaseSchemaUpdater` class found in the `Microsoft.Phone.Data.Linq` namespace. To create instances of `DatabaseSchemaUpdater`, an extension method named `CreateDatabaseSchemaUpdater` has been added to the `DataContext` class. The next listing details how to create and execute an update.

Listing 5.9 Upgrading an existing database

```
if (!db.DatabaseExists())
{
    db.CreateDatabase();
}
else
{
    var updater = db.CreateDatabaseSchemaUpdater();
    if (updater.DatabaseSchemaVersion == 0)
    {
        updater.AddColumn<HighScore>("Difficulty");
        updater.DatabaseSchemaVersion = 1;
        updater.Execute();
    }
}
```

① **Standard database creation**

② **Check schema version**

③ **Update schema and version**

The database upgrade code is added to the `HighScoreDatabaseRepository`’s constructor, in an `else` block following the check for database existence ①. The updater is created with a call to `CreateDatabaseSchemaUpdater`. Next you examine the current version of the database ②. When the database is at version zero, you know it was created by version 1.0 of the application and that you need to add the new column to the `HighScore` table. The column is added with the `AddColumn` method ③. You also change the database version to the value 1. When all the changes are complete, you commit the changes to the database with the `Execute` method. The `Execute` method applies all of the requested changes in a single transaction.

Debug the application and step through the database update code. If the database exists and is at version zero, the new column will be added to the HighScore table. If you exit the application and run it a second time, the database will be at version 1 and the upgrade code will be skipped.

There's a small problem with updating databases. The versioning scheme is an extension to LINQ to SQL specific to Windows Phone. The core LINQ to SQL code doesn't understand versioning, and databases are always created with a schema version of zero. As shown in figure 5.7, version 1.1 of your application also creates a database with a schema version of zero. A database upgraded by version 1.1 of the application will have a schema version of 1.

To demonstrate the problem, uninstall the HighScore sample application (or use the Rebuild Solution option in Visual Studio). The first time you run the application, the database will be created at version zero, but this time version 0 contains the `Difficulty` column. Exit the application and run it again. The database update code will try to execute but will generate an exception since the `Difficulty` column already exists.

How can you fix this problem? You can use the `DatabaseSchemaUpdater` to set the schema version immediately after you create the database. Before you change the version 1.1 code that creates and upgrades the database, back up and start the process over. Comment out the difficulty field in the `HighScore` class, and the `else` block with the upgrade code in the `HighScoreDatabaseRepository` constructor. Uninstall the application from the phone so that the current database is deleted. Deploy and run the code so that a database with schema 0 is created.

Now you're ready to fix the database create and update code in version 1.1. The new code is shown in the following listing.

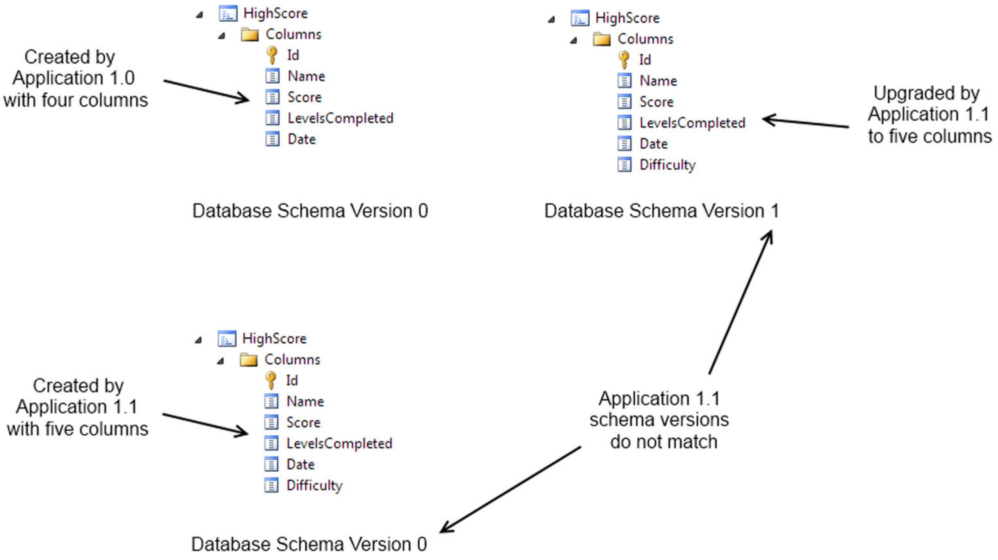

HighScore
Columns
Id
Name
Score
LevelsCompleted
Date
Created by Application 1.0 with four columns
Database Schema Version 0
HighScore
Columns
Id
Name
Score
LevelsCompleted
Date
Difficulty
Upgraded by Application 1.1 to five columns
Database Schema Version 1
HighScore
Columns
Id
Name
Score
LevelsCompleted
Date
Difficulty
Created by Application 1.1 with five columns
Application 1.1 schema versions do not match
Database Schema Version 0

Figure 5.7 Database schema as created and upgraded by the application

Listing 5.10 A better database upgrade implementation

```
if (!db.DatabaseExists())
{
    db.CreateDatabase();
    DatabaseSchemaUpdater updater = db.CreateDatabaseSchemaUpdater();
    updater.DatabaseSchemaVersion = 1;
    updater.Execute();
}
else
{
    DatabaseSchemaUpdater updater = db.CreateDatabaseSchemaUpdater();
    int databaseSchemaVersion = updater.DatabaseSchemaVersion;

    if (databaseSchemaVersion == 0)
    {
        updater.AddColumn<HighScore>("Difficulty");
        updater.DatabaseSchemaVersion = 1;
        updater.Execute();
    }
}
```

① **Newly created database is version 1**

② **Upgrade if database not version 1**

First you change the database creation code to record version 1 in `DatabaseSchemaVersion` ①. Finally, when the version of the schema is 0, you'll upgrade ② the database by adding the `Difficulty` column and changing the database version to 1.

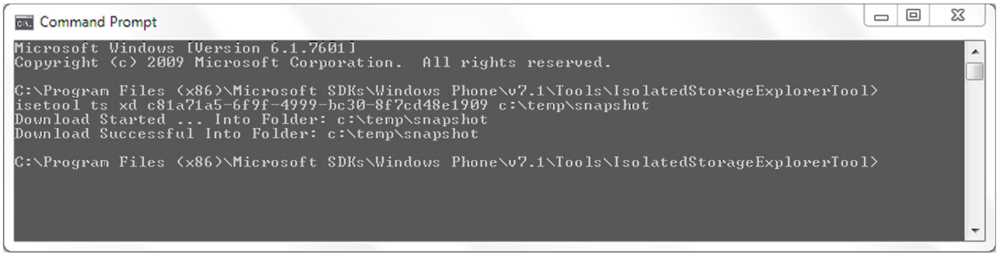
Run the application and your database with schema version 0 should be upgraded to schema version 1. Run the application a second time, and the update code should find the schema version of 1. Now uninstall the application and redeploy it. The first time you run the application after deploy, you create the database with a schema version of 1. The next time you run the application, the updater reports version 1 and the new code decides that there's no work to perform.

We're nearly finished with our coverage of LINQ to SQL. The one other feature of LINQ to SQL that's unique to Windows Phone revolves around deploying and using a read-only reference database.

5.4.8 *Adding a read-only database to your project*

Not all applications require writable databases and some databases are intended for read-only scenarios. Consider a database that contains ZIP codes and tax rates for a CRM application, or a dictionary of words for a word scramble game. Read-only databases can be included in your .xap deployment package in the same way that audio files, icons, and other content files can be included.

Read-only databases can be used directly from the application install folder and don't need to be copied into isolated storage. The connection string that's supplied to the `DataContext` allows you to specify the application installation folder as the location for a database file. How do you create the database file to begin with? One solution might be to create a Windows Phone project specifically to generate the database. The database generator project would share the data model classes along with the data context. The data model and data context



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool>
isetool ts xd c81a71a5-6f9f-4999-bc30-8f7cd48e1909 c:\temp\snapshot
Download Started ... Into Folder: c:\temp\snapshot
Download Successful Into Folder: c:\temp\snapshot

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool>
```

Figure 5.8 Taking a snapshot with the Isolated Storage Explorer

could be placed in a shared assembly project, or could be directly used in both projects using linked files.

The database generator project would call the data context's `CreateDatabase` method to create the database file. Once created, the database generator would create objects and insert them into the appropriate `Table` objects. Once the database is fully created, the application can exit.

For the next step you need the product ID from the generator project's WMAppManifext.xml file. The manifest file resides in the project's Properties folder. Find and open the file, and look for the Product ID:

```
<App xmlns="" ProductID="{c81a71a5-6f9f-4999-bc30-8f7cd48e1909}" …
```

Next fire up the Isolated Storage Explorer tool introduced in chapter 1. For this task you'll take a snapshot of isolated storage, which will copy all the files in isolated storage to a location on your computer's hard drive. Figure 5.8 shows the results of downloading a snapshot to the c:\temp\snapshot folder.

Once the snapshot is copied, you can find the SDF file and copy it into your project using the Visual Studio Project > Add Existing Item menu option. Mark the file's build action as Content and it'll be included in the application's .xap file.

Now when your application is deployed, the database will also be deployed. When the application is running, the reference database can be opened with the following connection string:

```
datasource='appdata:/file1.sdf';mode=read only
```

One advantage of a read-only database is apparent when working with a large dataset. A dictionary of words could easily be stored in an XML file and loaded into memory. Loading large datasets into memory becomes impractical, if not impossible, when the database size approaches 100 MB. Using LINQ to SQL, an application can query for and load only a subset of the database, reducing the memory footprint required by the application.

5.5 Summary

In this chapter we presented three solutions for storing application data in the Windows Phone filesystem. With this you now know how to store application data between runs of an application. Simple sets of data can be stored in application settings using `IsolatedStorageFile`. Documents, text files, and binary data can be written to files using `IsolatedStorageFile`. The last alternative involves relational databases, SQL Server CE, and LINQ to SQL.

LINQ to SQL is a broad subject and many topics require more detail than we can provide in our book. For more depth on LINQ to SQL see the MSDN documentation. There are also several books devoted entirely to the subject.

We also showed how to use the Isolated Storage Explorer tool to copy data from a phone, specifically for generating read-only databases. The Isolated Storage Explorer can also be used to load data into isolated storage, which is ideal for restoring a test device to a known state, allowing for the execution of specific test cases.

The data storage concepts discussed in this chapter should be familiar to .NET Framework developers, as they're limited versions of the same storage technologies that have existed in the .NET Framework for many years. In the next chapter you'll learn how to use the `PhotoChooserTask`, the `CameraCaptureTask`, and the `PhotoCamera` classes to develop camera-based applications.

6

Working with the camera

This chapter covers

- Working with the photo and camera choosers
- Taking pictures with the camera hardware
- Editing pictures
- Saving pictures to isolated storage

You're reading this book because you want to build mobile applications. In order to build a useful mobile application, you need to understand how most mobile users use their phones. Are they only using their phone for talking and sending text messages? No. A lot of mobile users are using more than just a phone. They use their device for listening to music or the radio, watching videos, and taking photos. As mobile developers, we need to know how to integrate multimedia in our applications, allowing users to do what they want to do in better and more efficient ways. In this chapter you're going to learn how to use the camera to capture photographs and include them in your application.

In this chapter you learn how to integrate the phone camera into your application. You may remember that we skipped the `PhotoChooserTask` and `CameraCaptureTask` in chapter 4, when we covered the other launchers and choosers. Both of these tasks are related to the camera and we're going to explain them in detail in this chapter. The `PhotoChooserTask` allows your application to prompt the user to pick a picture from the Pictures Hub to load into the application. The `CameraCaptureTask` allows your application to use the built-in camera application to take a picture for use in your application.

Working with the built-in camera application is great, but some applications need access to the underlying camera hardware and video stream. The Windows Phone SDK 7.1 introduced a new `PhotoCamera` API enabling advanced camera operation. In this chapter you'll build a sample application demonstrating how to use both choosers and the `PhotoCamera` API to capture a picture. Once the picture is captured, you'll add a simple image editing feature and save the picture to isolated storage.

Let's get started with the first sample application, which you'll name PhotoEditor.

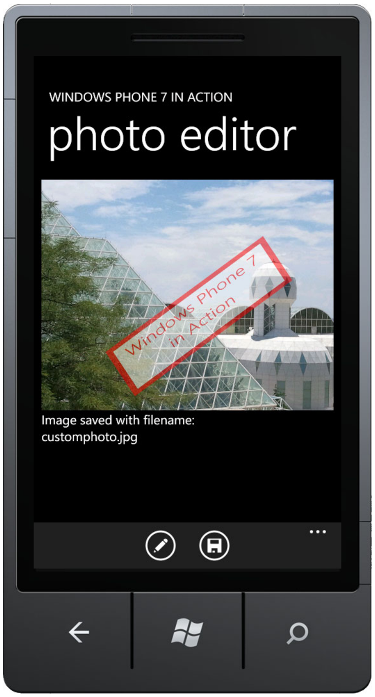


Image saved with filename:
customphoto.jpg

Figure 6.1 The PhotoEditor sample application

6.1 Starting the PhotoEditor project

The PhotoEditor will capture a photograph using several different APIs. The user will be able to pick a photograph from their media library using the `PhotoChooserTask`. The user will also be able to take a picture using the built-in camera application with the `CameraCaptureTask`. Next we'll show you how to use the `PhotoCamera` API to turn the PhotoEditor into a custom camera application.

After you learn how to load or capture pictures, you'll put the *edit* in PhotoEditor. You'll implement a simple feature that places a stamp on a photograph. The sample application with a stamp can be seen in the screenshot displayed in figure 6.1.

After stamping a picture, the user will be able to save the edited picture to isolated storage. In addition to routines to save the picture, you'll also create routines to reload the picture from a file in isolated storage.

You start the PhotoEditor application by creating a new project from the Windows Phone Application project template. Name the project PhotoEditor. Once the project is created, open the MainPage.xaml file and add the following markup to the `ContentPanel` grid:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="360" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Rectangle x:Name="photoContainer" Fill="Gray" />
    <TextBlock x:Name="imageDetails" Grid.Row="1" TextWrapping="Wrap"
        Text="Choose an image source from the menu."/>
</Grid>
```

The `ContentPanel` is dived into two rows. The first row contains a `Rectangle` control named `photoContainer`. You'll render the selected picture image in this rectangle using an `ImageBrush`. Later you'll use the rectangle to display the live stream from the `PhotoCamera`. The second row contains a `TextBlock` that's used to provide the user with instructions, information about the image, and error messages. The second row is automatically sized, taking up all the space not consumed by the 360-pixel-high first row.

Now that you have the skeleton of the project ready, let's discuss how to load a picture into the `photoContainer` `Rectangle` using one of the two camera tasks.

6.2 Working with the camera tasks

All Windows Phones have at least one built-in camera. Microsoft's minimum hardware specification requires a camera with a flash. Some phones have a second front-facing camera. The Windows Phone class library contains two chooser tasks that provide limited access to the camera, or more specifically, to pictures taken by the camera or that exist in the media library.

> **NOTE** Only photographs are accessible via the chooser tasks. The chooser classes don't allow you to access videos from the camera.

The chooser tasks are the `PhotoChooserTask` and `CameraCaptureTask`. Let's look at choosing from the media library with the `PhotoChooserTask` first.

6.2.1 Choosing a photo with PhotoChooserTask

All phone users are familiar with how to take a picture with their mobile device, since taking pictures is a part of their daily phone usage. But as programmers, knowing how to take a photo with the phone isn't enough. The `PhotoChooserTask` allows an application to launch the built-in Photo Chooser application.

You need to know how to use the camera from your code and how to integrate with your custom application. This is what you're going to learn in this section. `PhotoChooserTask` allows users to select the photo from phone memory with the built-in Photo Chooser. The Photo Chooser has a navigation button that can take you to the camera application. The `PhotoChooserTask` is handy for those applications that allow a user to choose a photograph and use it somewhere in the application. One example is adding a picture to a user's profile.

If you're coming from a .NET background then you might be wondering about the differences between `OpenFileDialog` and `PhotoChooserTask` when they both select files. The Windows Phone APIs don't give you a generic file picker so you'll have to use `PhotoChooserTask` in order to allow the user to select a photo in your application. Table 6.1 describes the members of `PhotoChooserTask`.

Table 6.1 PhotoChooserTask members

Name	Type	Description
`Show`	Method	Shows the Phone Chooser application
`ShowCamera`	Property	Gets or sets whether the user is presented with a button for launching the camera during the photo choosing process
`Completed`	Event	Occurs when a chooser task is completed

`PhotoChooserTask`'s `Show` method launches the built-in Photo Chooser application. When the `Show` method executes, your application is moved to the background as the operating system switches to the Photo Chooser application. The Photo Chooser application is shown in figure 6.2. The Photo Chooser opens to a list of albums that the user has created in the phone. There are only two default albums named Camera Roll and Sample Pictures in the emulator but the user might have more in a real phone device. Note that Camera Roll is the default album that stores all photos taken by using the phone camera.

The `ShowCamera` property can be set to `true` if the user should see the camera icon in the application bar. When the camera icon is shown, the user has the option to take a new picture. It's always a best practice to set the `ShowCamera` property to `true`, because the user may want to take a new photo right away instead of selecting one from the photo albums.

You'll add an `ApplicationBar` menu item to show the `PhotoChooserTask` in your `PhotoEditor` application:

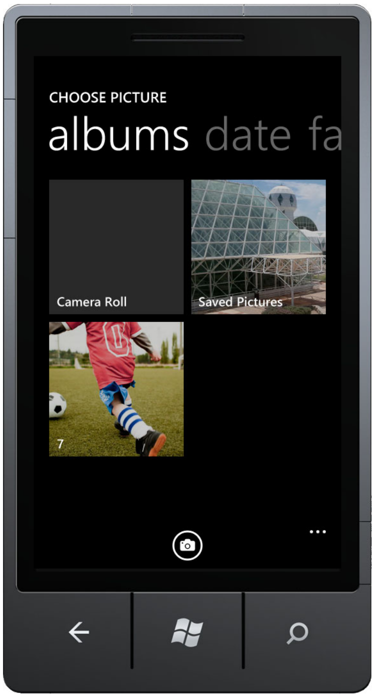


Figure 6.2 A screenshot of the built-in Photo Chooser application with camera button

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBar.MenuItems>
            <shell:ApplicationBarMenuItem Text="choose photo"
                Click="Choose_Click" />
        </shell:ApplicationBar.MenuItems>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

The `ApplicationBarMenuItem` displays the text "choose photo". The click event handler for the menu item constructs a `PhotoChooserTask` and sets the `ShowCamera` property to true:

```
private void Choose_Click(object sender, EventArgs e)
{
    var task = new PhotoChooserTask();
    task.ShowCamera = true;
    task.Completed += chooserTask_Completed;
    task.Show();
}
```

The `Completed` event is wired up to the `chooserTask_Completed` event handler. You may recall from chapter 4 that when the `Show` method is called, your application is paused and placed in the Task Switcher's back stack. When the user completes the chooser operation, the operating system navigates back to your application. The `Completed` event handler is called when the application resumes operation in the foreground.

> **NOTE** The media library is locked when a device is connected to a computer and the Zune software is running. When the media library is locked you'll encounter errors when testing the PhotoChooserTask. Review the use of the WPConnect tool covered in chapter 1.

The `Completed` event will be raised when the user selects a photo from the photo library, takes a new photo, or uses the Back button to navigate back to the application. The following listing shows how to handle the `Completed` event and display the selected photo in the image control.

Listing 6.1 Displaying the selected photograph

```
private WriteableBitmap currentImage;

void chooserTask_Completed(object sender, PhotoResult e)
{
    if (e.TaskResult == TaskResult.OK)
    {
        currentImage = PictureDecoder.DecodeJpeg(e.ChosenPhoto);

        photoContainer.Fill = new
            ImageBrush{ ImageSource = currentImage };

        imageDetails.Text = string.Format("Image from {0}\n",
            sender.GetType().Name);
        imageDetails.Text += string.Format("Original filename:\n{0}",
            e.OriginalFileName);
    }
    else
    {
        photoContainer.Fill = new SolidColorBrush(Colors.Gray);
        imageDetails.Text = e.TaskResult.ToString();
    }
}
```

① **Field to hold chosen photo**

② **Photo stream in e.ChosenPhoto**

③ **Display image with brush**

First you need a new `WriteableBitmap` field named `currentImage` ① in which you store the photograph shown in the `photoContainer Rectangle` control. It's a best practice to check the `TaskResult` property to determine whether the chooser task completed `OK` or whether the task was canceled. The `PhotoResult` class defines two properties named `ChosenPhoto` and `OriginalFileName`. `ChosenPhoto` contains the selected picture's data in a `Stream`. You pass the image stream to the `DecodeJpeg` method of the helper class named `PictureDecoder` ❷. The `PictureDecoder` class is found in the `Microsoft.Phone` namespace. The `DecodeJpeg` method builds a `WriteableBitmap`. You use the `WriteableBitmap` as the source for an `ImageBrush` and the brush is used as the `Fill` for the `photoContainer` ❸. Finally you build a message informing the user that the picture came from the `PhotoChooserTask` and had the filename specified in the `OriginalFileName` property.

When the `TaskResult` isn't `OK`, you clear the `photoContainer` by setting the `Fill` property to a `SolidColorBrush` and display the `TaskResult` value in the `imageDetails` `TextBlock`.

Run the application, select the choose photo menu item, and pick a photo. The photo you choose should be shown in the application when the `PhotoChooserTask` completes. Next we look at how to use the `CameraCaptureTask`.

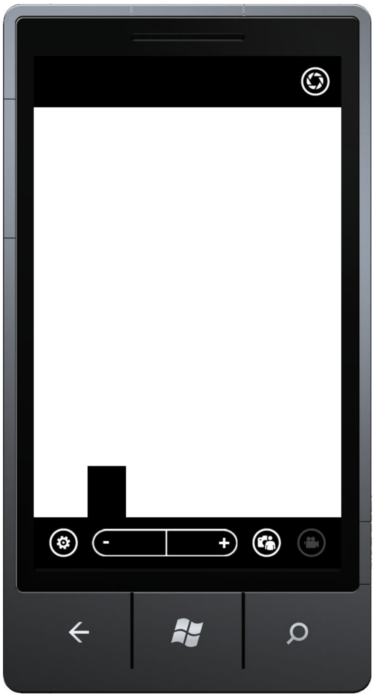


Figure 6.3 The camera application in the emulator

6.2.2 Taking photos with CameraCaptureTask

`CameraCaptureTask` is another useful task that allows developers to launch the built-in camera application. Unlike `PhotoChooserTask`, it allows you to open the camera directly from an application without requiring the user to perform additional steps. Shown in figure 6.3, the camera application in the emulator shows a placeholder white screen with a moving black block. You can take a photo in the emulator by clicking the round widget located at the right top corner. We recommend you use a physical device to test this task.

The `CameraCaptureTask` doesn't have any properties, and only implements the `Show` method. Show the camera application by calling the `Show` method from the click event of a new menu item you add to MainPage.xaml:

```
<shell:ApplicationBarMenuItem Text="capture photo" Click="Capture_Click" />
```

The following snippet shows how to use `CameraCaptureTask`:

```
private void Capture_Click(object sender, EventArgs e)
{
    var task = new CameraCaptureTask();
    task.Completed += chooserTask_Completed;
    task.Show();
}
```

You subscribe to the `Completed` event to receive the photo stream from the camera and call the `Show` method to launch the camera. For the `Completed` event, you can use the same event handler method that you used in `PhotoChooserTask` sample.

Deploy the updated application to your phone, run the application, and choose the capture photo menu option. Hold the phone in the landscape orientation, snap a picture, and press the Accept button. When the task completes, you should see your new photo displayed in the `photoContainer`. Select the capture photo menu option again, but this time hold your phone in the portrait orientation when you snap the picture. When the task returns this time, the new picture is displayed sideways. The important thing to note is that the image stream that returns from the camera won't respect the orientation of your application. We show you how to handle the orientation of a picture in the next section.

6.2.3 Handling picture orientation in CameraCaptureTask

Please take a look at figure 6.4. Do you notice something wrong? The orientation of the photograph doesn't match the orientation of the application.

The photo stream that returns from the camera doesn't know or care about the orientation of your application. The picture's orientation will be based on how the user holds the phone while taking the picture. You can't ask your user to hold the photo in a specific position; they'll hold the phone however they like. All you can do is adjust the orientation of the photo before showing it in your application.

> **TIP** The image rotation techniques shown in this section were inspired by Microsoft blogger Tim Heuer, who wrote a good post about orientation mismatch in his blog. You can read his blog post at http://mng.bz/fk44.

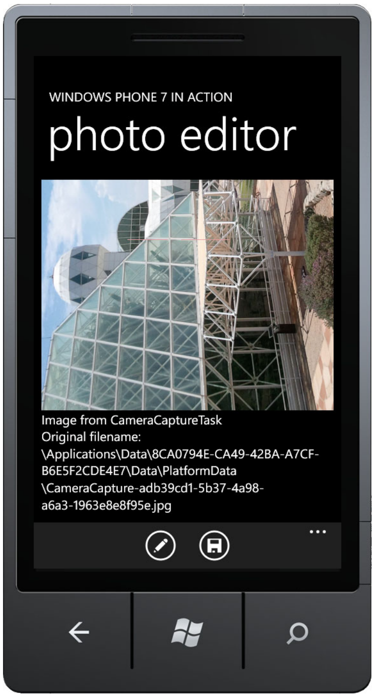


Image from CameraCaptureTask
Original filename:
\Applications\Data\8CA0794E-CA49-42BA-A7CF-B6E5F2CDE4E7\Data\PlatformData\CameraCapture-adb39cd1-5b37-4a98-a6a3-1963e8e8f95e.jpg

Figure 6.4 Orientation problems with CameraCaptureTask

The `CameraCaptureTask` writes metadata about the picture to the stream along with the image data. The image data and metadata are in the *Exchange Image File Format* (EXIF). EXIF has an attribute that you can read to determine the orientation of the photo. The `PhotoResult`'s `ChosenPhoto` property is a `Stream` and you need a way to extract the EXIF data from the stream. In this example, you're going to use ExifLib, an open source EXIF library that can be downloaded from http://mng.bz/UW7A. We've included the ExifLib assembly in the sample project. Feel free to use an EXIF library of your choice or you can even roll your own library.

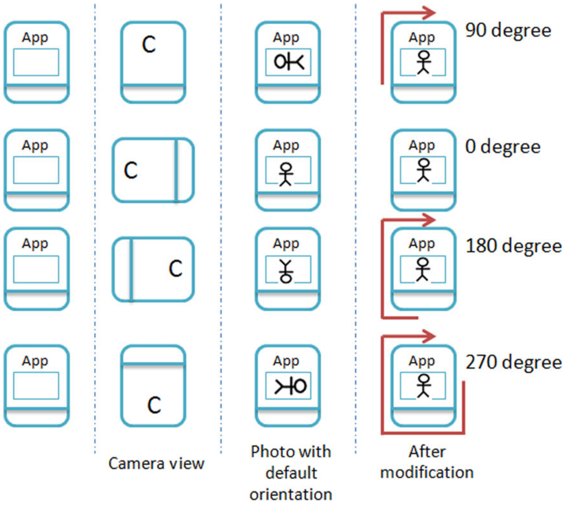


Figure 6.5 Different ways of holding the camera, photo output, and degree that the picture needs to be rotated

Before you think about how to modify the orientation, you need to figure out the default orientation of the photo that you get from the camera and how the orientation is different from the application's orientation. Figure 6.5 shows the possible combinations of application and camera orientation.

This figure shows four different ways of holding the camera and the output in a clear manner. The first column of this diagram illustrates that your sample application is always in a portrait orientation. The second column shows the different ways of holding the camera. The third column displays the output photo from the camera. As you can see, the default orientation of a photo will be different based on the way that the user holds the camera. The last column shows how many degrees you need to rotate the photo in order to get an orientation that matches the application.

The code that performs the desired modifications shown in figure 6.5 is implemented in two steps. The first step calculates the angle to rotate the image, and the second step performs the rotation. The next listing demonstrates how to calculate the degree of rotation in the `GetAngleFromExif` method.

Listing 6.2 Calculate how many degrees to rotate a photo

```
using ExifLib;
using System.IO;

int GetAngleFromExif(Stream imageStream)
{
    var position = imageStream.Position;
    imageStream.Position = 0;
    ExifOrientation orientation = ExifReader.ReadJpeg(
        imageStream, String.Empty).Orientation;
    imageStream.Position = position;

    switch (orientation)
    {
```

Use EXIF library

❶ **Read orientation**

❷ **Determine rotation angle**

```
        case ExifOrientation.TopRight:
            return 90;
        case ExifOrientation.BottomRight:
            return 180;
        case ExifOrientation.BottomLeft:
            return 270;
        case ExifOrientation.TopLeft:
        case ExifOrientation.Undefined:
        default:
            return 0;
    }
}
```

First you read the orientation from the EXIF formatted stream. The `ExifReader.ReadJpeg` method accepts an image stream and returns a `JpegInfo` instance. The `JpegInfo` object contains information stored in the EXIF header in the image stream, including its orientation. You extract the orientation, in the form of an `ExifOrientation` instance, from the `Orientation` property ❶. Based on that, you'll define how many degrees you need to rotate ❷. We suggest that you compare the implementation and the diagram in figure 6.5 to understand more details.

Once you know how many degrees you need to rotate the original photo, it's easy to create a new photo with the correct orientation based on the original photo. The `WriteableBitmap` class provides direct access to the pixel level information of bitmaps. The image data is accessed through the `Pixels` property, a one-dimensional array of pixels laid out in a row-first pattern. The `PixelWidth` and `PixelHeight` properties are used to determine how many rows and columns of pixels exist in the two-dimensional image. The first row of pixels resides in the `Pixels` array from index 0 to `PixelWidth`-1. The second row of the images resides in the `Pixels` array from index `PixelWidth` to 2 * `PixelWidth` -1. The following listing shows how to read the pixels from the original photo and copy them to the right position in the new bitmap image.

Listing 6.3 Copy the pixels from original bitmap to new bitmap

```
private WriteableBitmap RotateBitmap(WriteableBitmap source,
    int width, int height, int angle)
{
    var target = new WriteableBitmap(width, height);
    int sourceIndex = 0;
    int targetIndex = 0;
    for (int x = 0; x < source.PixelWidth; x++)
    {
        for (int y = 0; y < source.PixelHeight; y++)
        {
            sourceIndex = x + y * source.PixelWidth;
            switch (angle)
            {
                case 90:
                    targetIndex = (source.PixelHeight - y - 1)
                        + x * target.PixelWidth;
                    break;
```

❶ **Create empty bitmap**

❷ **Determine target location**

```
                case 180:
                    targetIndex = (source.PixelWidth - x - 1)
                        + (source.PixelHeight - y - 1) * source.PixelWidth;
                    break;
                case 270:
                    targetIndex = y + (source.PixelWidth - x - 1)
                        * target.PixelWidth;
                    break;
            }
            target.Pixels[targetIndex] = source.Pixels[sourceIndex];
        }
    }
    return target;
}
```

Copy pixels ❸

The `RotateBitmap` method accepts four parameters: the source bitmap, the width and height of the new bitmap, and the angle the bitmap should be rotated. The method starts off creating a new `WriteableBitmap` with the specified width and height ❶. You loop through each pixel in the source bitmap, calculating the index of the source pixel and the index in the target bitmap where you'll copy the pixel. Depending on the rotation angle ❷, you calculate the target index using one of three different expressions. Finally, you copy the pixel ❸ from the source bitmap to the target bitmap.

> **NOTE** If you'd like more information about `WriteableBitmap`, please read the MSDN documentation for Silverlight's version of WriteableBitmap: http://mng.bz/XCzN.

`RotateBitmap` accepts a `WriteableBitmap`. The `CameraOperationCompletedEventArgs` passed to the `camera_CaptureCompleted` method provides a `Stream`. Currently the `camera_CaptureCompleted` method uses `DecodeJpeg` to convert the stream to a `WriteableBitmap`. You'll wrap the call to the `DecodeJpeg` and `RotateBitmap` methods in a new `DecodeImage` method. The code for the new method is shown next.

Listing 6.4 Decoding the image stream

```
private WriteableBitmap DecodeImage(Stream imageStream, int angle)
{
    WriteableBitmap source = PictureDecoder.DecodeJpeg(imageStream);

    switch(angle)
    {
        case 90:
        case 270:
            return RotateBitmap(source, source.PixelHeight,
                source.PixelWidth, angle);
        case 180:
            return RotateBitmap(source, source.PixelWidth,
                source.PixelHeight, angle);
        default:
            return source;
    }
}
```

❶ **Swap width and height**

❷ **Skip RotateBitmap**

The new `DecodeImage` method accepts an image stream and the rotation angle. The first step is to convert the image stream into a `WriteableBitmap` using the `PictureDecoder` helper class. Next you examine the degree of rotation, calling `RotateBitmap` if necessary. When you're rotating the image by 90 or 270 degrees, you're swapping the width and height ❶. When the angle is 0, you don't call `RotateBitmap` ❷.

Using the `GetAngleFromExif` you wrote in the last section along with the new `DecodeImage` method, you can update the `chooserTask_Completed` method to rotate the captured photos. In the completed event handler, replace the call to `PictureDecoder.DecodeJpeg` with the following snippet:

```
int angle = GetAngleFromExif(e.ChosenPhoto);
currentImage = DecodeImage(e.ChosenPhoto, angle);
```

You now have a good understanding about how `PhotoChooserTask` and `CameraCaptureTask` work. You've also learned how to handle the picture orientation. These chooser tasks are useful when choosing a picture is a secondary feature of your application. When controlling the camera is a central feature of your application, you want to use the `PhotoCamera` API.

6.3 Controlling the camera

When Windows Phone was first released, the only way to access the camera was with the `CameraCaptureTask`. The Windows Phone SDK 7.1 introduced APIs allowing programmatic access to the camera hardware. The `PhotoCamera` class allows a developer access to the image data as it's captured by the camera hardware. The `CameraButtons` classes allow applications to detect when the user has pressed or released the shutter button built into every phone.

> **TIP** Applications that use the `PhotoCamera` class must include the `ID_CAP_ISV_CAMERA` capability in their application manifest. You should include `ID_HW_FRONTCAMERA` to access the front-facing camera. As we mentioned in chapter 1, the front camera capability isn't automatically added to the manifest when the project is generated by Visual Studio.

Every Windows Phone has a built-in primary camera. Some Windows Phones also have a front-facing camera. `PhotoCamera` supports both cameras, allowing the developer to specify which camera to use when constructing an instance of `PhotoCamera`. The camera is specified with either the `Primary` or `FrontFacing` values defined in the `CameraType` enumeration. The following snippet demonstrates how to construct a `PhotoCamera` that reads from the front-facing camera:

```
if(PhotoCamera.IsCameraTypeSupported(CameraType.FrontFacing))
    camera = new PhotoCamera(CameraType.FrontFacing);
```

The `PhotoCamera` class provides a static method named `IsCameraTypeSupported` which is used to determine whether the phone has a front-facing camera. Front-facing cameras, also called *self-portrait cameras*, didn't appear until Windows Phone 7.5

devices started shipping in the fall of 2011. Developers should check whether a front-facing camera is installed before trying to access it.

The `PhotoCamera` class provides access to the live preview image picked up by the camera lens. The camera buffers the preview image into arrays of raw image data. The preview buffer is accessed via three different methods, each providing the raw image data in a different format:

- `GetPreviewBufferArgb32` returns image data in the same 4-byte ARGB structure used by Silverlight.
- Luminance and chrominance information, as defined by the YCbCr color model, is returned through the `GetPreviewBufferYCbCr` method.
- The `GetPreviewBufferY` method returns just Luminance information.

The `GetPreviewBuffer` methods are useful for a variety of real-time image processing and augmented reality applications. Your sample application won't perform any sophisticated image processing and displays an unaltered preview image prior to capturing the image when the user presses the camera button. Microsoft included new APIs designed for just this scenario, which we'll demonstrate in the next section.

You need to perform a few steps to prepare the PhotoEditor application to use the `PhotoCamera`. You need to add a new menu item allowing the user to pick the live camera as the source of an image. You also need to add a click event handler responsible for creating and initializing a `PhotoCamera`. First you add the menu item to the application bar declared in MainPage.xaml.cs:

```
<shell:ApplicationBarMenuItem Text="custom camera" Click="Camera_Click" />
```

When the user presses the custom camera menu item, the application will show a live preview image in the `photoContainer`. The preview image will continue to be shown until the user either presses the menu item a second time, or captures a snapshot using the phone's camera shutter button. You'll start with the `Click` event handler which is shown next.

Listing 6.5 The camera click event handler

```
PhotoCamera camera;                                          ❶ Class level field

private void Camera_Click(object sender, EventArgs e)
{
    if (camera == null)                                      ❷ Is camera already in use?
    {
        currentImage = null;
        imageDetails.Text = string.Format("Choose custom camera again to
➥  close camera. Use the hardware shutter button to take a picture.\n");
        InitializeCamera();
    }
    else
    {
        CleanUpCamera();
        photoContainer.Fill = new SolidColorBrush(Colors.Gray);
        imageDetails.Text = "Choose an image source from the menu.";
    }
}
```

You need a new class-level field to reference the instance of the `PhotoCamera` ❶ you create when the user clicks the custom camera menu item for the first time. If the field is null, you prepare the application by clearing the current image, displaying a message to the user, and initializing the camera. If the camera is already in use ❷, the user has tapped the menu item a second time to cancel the custom camera operation, and you clean up the camera and clear the `photoContainer`. We'll look at the `CleanUpCamera` method later, but let's look at `InitializeCamera` now. You might want to create empty placeholder methods in your code so that the project compiles.

The `InitializeCamera` method is where you hook up interesting events and capture the preview buffer so you can display a viewfinder to the user. You'll enhance the `InitializeCamera` method in later sections, but you start the method's implementation here by constructing a new instance of `PhotoCamera`:

```
void InitializeCamera()
{
    camera = new PhotoCamera(CameraType.Primary);
    camera.Initialized += camera_Initialized;
}
```

The first camera-related event you wire up is the `Initialized` event. A `PhotoCamera` instance isn't fully ready for use when the constructor completes. The `PhotoCamera` class raises the `Initialized` event once it's completely ready.

A fully initialized `PhotoCamera` is able to report which resolutions are supported when previewing images and capturing pictures. Supported resolutions are reported in the `AvailableResolutions` property. A developer can determine the resolution of the preview buffer with the `PreviewResolution` property. The `Resolution` property not only reports the current resolution, but allows the developer to specify which of the available resolutions should be used to capture a photograph.

You'll display the `PreviewResolution` and `Resolution` values to the user once the camera is initialized. The next listing implements the `camera_Initialized` event handler.

Listing 6.6 Reading supported resolutions in the `Initialized` event

```
void camera_Initialized(object sender, CameraOperationCompletedEventArgs e)
{
    Dispatcher.BeginInvoke(() =>                    ❶ Update on UI thread
    {
        imageDetails.Text += string.Format("{0} supported resolutions.\n",
            camera.AvailableResolutions.Count());
        imageDetails.Text += string.Format("Current resolution: {0}\n",
            camera.Resolution);
        imageDetails.Text += string.Format("Preview resolution: {0}\n",
            camera.PreviewResolution);
    });
    camera.Initialized -= camera_Initialized;          ❷ Unwire event
}
```

The `Initialized` event handler isn't called on the UI thread. Because your code updates the user interface, you use the `Dispatcher` ❶ to execute your code on the UI thread. After updating the user interface, you don't need to capture `Initialized` events and unsubscribe the event handler ❷.

Now that you know about the `PhotoCamera`, how can you use it in your PhotoEditor application? You already display pictures in the `photoContainer Rectangle` using an `ImageBrush`. You can turn the `photoContainer` into a camera view finder using a `VideoBrush`.

6.3.1 Painting with the VideoBrush

The `VideoBrush` class in an implementation of the `Brush` class and is designed to paint video instead of a solid color, gradient color, or image. The `VideoBrush` first appeared in Silverlight 3 for the browser. The Windows Phone SDK 7.1 marks the first appearance of the `VideoBrush` for the phone. The SDK also introduced an extension method named `SetSource` to enable painting the camera's preview buffer with a `VideoBrush`. To see a `PhotoCamera`-driven `VideoBrush` in action, add the following code to the end of the `InitializeCamera` method:

```
var brush = new VideoBrush();
brush.SetSource(camera);
brush.RelativeTransform = new RotateTransform
    { CenterX = 0.5, CenterY = 0.5, Angle = camera.Orientation };
photoContainer.Fill = brush;
```

The `SetSource` extension method used here is defined in the `CameraVideoBrushExtensions` class and is found in the `Microsoft.Devices` namespace.

You learned earlier in the chapter that the camera's orientation doesn't always match the application's orientation. You solved the orientation problem with the `CameraCaptureTask` by rotating the captured picture. To solve the orientation problem with the preview buffer, you rotate the `VideoBrush` using a `RotateTransform`. The camera tells you the amount of mismatch with the `Orientation` property, which you use as the source of the `RotateTransform`'s `Angle` property.

> **TIP** `RotateTransform` is just one of several transform classes that can be used to manipulate Silverlight elements and controls. If you're unfamiliar with Silverlight's transform classes, you can read more about them in the book *Silverlight 5 in Action* by Pete Brown or from the MSDN article titled "Transforms."

Run the application now and select the custom camera option. You should see the photo container turn into a view finder. Pick up the phone and move it around. The image in the viewfinder updates in real time. Now all you need is the ability to capture the picture.

6.3.2 Snapping a photo

When using the built-in camera application, the user presses the camera shutter button to capture a picture. Applications that use the `PhotoCamera` to capture pictures should also use the camera shutter button. The camera shutter button is exposed to the developer with events on the static `CameraButtons` class.

Three events are provided by the `CameraButtons` class named `ShutterKeyHalfPressed`, `ShutterKeyPressed`, and `ShutterKeyReleased`. In the built-in camera application, a half press of the shutter key focuses the camera and a full press snaps the picture. You'll use the `ShutterKeyPressed` event in your application as the trigger to capturing a picture. Add the following line of code to the `InitializeCamera` method, right before the `VideoBrush` code you added in the last section:

```
CameraButtons.ShutterKeyPressed += cameraButtons_ShutterKeyPressed;
```

The `ShutterKeyPressed` event will be handled by the `cameraButtons_ShutterKeyPressed` method. The only responsibility of the event handler is to start the process that snaps the picture and captures the image in a JPEG image stream. The captures process is started with a call to `PhotoCamera`'s `CaptureImage` method:

```
private void cameraButtons_ShutterKeyPressed(object sender, EventArgs e)
{
    camera.CaptureImage();
}
```

The process that captures the picture raises four events named `CaptureStarted`, `CaptureImageAvailable`, `CaptureThumbnailAvailable`, and `CaptureCompleted`. `CaptureStarted` is raised first, and `CaptureCompleted` is raised once the capture process has completed. In between these two events, `CaptureImageAvailable` and `CaptureThumbnailAvailable` are raised, providing the calling application the opportunity to save either a full resolution or thumbnail copy of the captured image.

The PhotoEditor application is only interested in the `CaptureImageAvailable` and `CaptureCompleted` events. Wire up two new event handlers in the `InitializeCamera` method:

```
camera.CaptureImageAvailable += camera_CaptureImageAvailable;
camera.CaptureCompleted += camera_CaptureCompleted;
```

First we're going to look at the `CaptureImageAvailable` event handler which you'll name `camera_CaptureImageAvailable`. The event handler is passed an instance of the `ContentReadyEventArgs` class, which provides the image stream containing the image captured by the camera:

```
void camera_CaptureImageAvailable(object sender, ContentReadyEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        currentImage = DecodeImage(e.ImageStream, (int)camera.Orientation);
        photoContainer.Fill = new ImageBrush{ ImageSource = currentImage };
        imageDetails.Text = "Image captured from PhotoCamera.";
    });
}
```

The `CaptureImageAvailable` event isn't raised on the UI thread so you use the `Dispatcher` to execute your user interface update code on the UI thread. You decode the image stream by calling the `DecodeImage` method you created earlier in the chapter. You use the camera's `Orientation` property as the rotation angle. The captured image is displayed in the `photoContainer` control.

When the picture capture process is complete you need to clean up the camera. You perform the camera cleanup in the `CaptureCompleted` event handler, which is shown in the next listing.

Listing 6.7 Checking for errors in the `CaptureCompleted` event handler

```
void camera_CaptureCompleted(object sender,
    CameraOperationCompletedEventArgs e)
{
    if (!e.Succeeded)
    {
        photoContainer.Fill = new SolidColorBrush(Colors.Gray);
        imageDetails.Text = "Camera capture failed.\n"
            + e.Exception.Message;
    }
    CleanUpCamera();
}
```

❶ **Check for errors**

❷ **Display exception's message**

An instance of the `CameraOperationCompletedEventArgs` class is passed into the event handler. Your code checks the `Succeeded` property ❶ to determine whether the image capture process completed as expected. If the operation failed, you clear the `photoContainer` and display the message ❷ from the `Exception` property.

In the PhotoEditor application, you only use the `PhotoCamera` when the user has selected the custom camera menu option. Once the user has snapped a picture, you need to clean up all registered event handlers and dispose of the camera:

```
void CleanUpCamera()
{
    CameraButtons.ShutterKeyPressed -= cameraButtons_ShutterKeyPressed;
    camera.CaptureImageAvailable -= camera_CaptureImageAvailable;
    camera.CaptureCompleted -= camera_CaptureCompleted;
    camera.Dispose();
    camera = null;
}
```

The `CleanUpCamera` method nearly completes your custom camera feature. Run the application, choose the custom camera, and snap a picture. You should see the picture ready for use shortly after pressing the shutter button. Now try one more thing—launch the application, and select the custom camera menu option. With the camera enabled and the viewfinder active, press the Start button to switch to the start screen, then use the Back button to navigate back. You might notice that the viewfinder is no longer active when returning from fast application switching.

6.3.3 Supporting fast application switching

Developers that use the `PhotoCamera` need to understand how fast application switching impacts their application code. Because the camera hardware is a shared resource, the operating system disconnects the camera when an application is sent to the background in favor of another task. When the application is restored to the foreground, the camera isn't automatically restored.

Camera-based applications should clean up camera resources when they're moved into the background. In chapter 3, you learned that an application can detect when it's entering a dormant state with the `Deactivated` event or in the `OnNavigatedFrom` override in their user interface pages. The PhotoEditor application will use the `OnNavigatedFrom` method:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    if (camera != null)
    {
        CleanUpCamera();
        State["customCamera"] = true;
    }
}
```

The PhotoEditor only uses the camera in the scenario when the user has selected the custom camera menu option. In this situation, the camera field isn't `null`, and you call the `CleanUpCamera` method to unwire the event handlers and dispose of the camera. You also add a `customCamera` key to the page's `State` dictionary as a flag you can look at when the application is restarted.

The `OnNavigatedTo` method is called once the application is restarted and `MainPage` is reloaded. Your code looks for the `customCamera` flag in the `State` dictionary. If the flag is found, you remove the flag and initialize the camera:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if(State.ContainsKey("customCamera"))
    {
        State.Remove("customCamera");
        InitializeCamera();
    }
}
```

Run the application and perform the application switching steps you used earlier. This time, the viewfinder should be reactivated when switching back to the PhotoEditor. Speaking of editing, you haven't actually implemented any editing features. Let's look at one now.

6.4 Image editing

The PhotoEditor application edits pictures by stamping them with the text "Windows Phone 7 in Action" surrounded by a border. The stamp can be seen in figure 6.1, which is the screenshot we showed you at the beginning of the chapter. It's not much fun to edit an image unless you can also keep the changes, so you're going to allow the user to save the stamped picture to a file.

The two new features are accessed with application bar buttons added to MainPage.xaml to enable the editing and saving of images you load or capture within the application:

```
<shell:ApplicationBarIconButton IconUri="/Images/appbar.edit.rest.png"
    Text="edit" Click="Edit_Click" />
<shell:ApplicationBarIconButton IconUri="/Images/appbar.save.rest.png"
    Text="save" Click="Save_Click" />
```

The images for the buttons come from the icons library installed with the Windows Phone SDK.

To implement the stamp, you might think you need to design a complex algorithm to access the raw image data and alter the appropriate pixel values. Instead you're going to leverage a feature of the Silverlight Framework that will draw a Silverlight `UIElement` over the top of the image contained within the `WriteableBitmap`. This feature allows you to define the stamp using a `TextBlock` and a `Border` control inside MainPage.xaml:

```
<Border x:Name="photoStamp" Height="125" Width="300" Opacity="0.5"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Background="White" BorderBrush="Red" BorderThickness="10">
    <TextBlock Text="Windows Phone 7 in Action" Foreground="Red"
        Style="{StaticResource PhoneTextLargeStyle}"
        TextWrapping="Wrap" TextAlignment="Center" />
</Border>
```

The `Border` is named `photoStamp` and defined with a `BorderThickness` of 10. The `Border` is drawn with a red outline and a white background. Inside the `Border` is a `TextBlock` that draws red text. The `Border` is given an `Opacity` of 0.5 so that the picture shows through the stamp. The `Grid` control draws elements in the order they're added in XAML and you want to make sure you add the `photoStamp` element as the first child of the `ContentPanel` so that it's hidden by the `photoContainer` `Rectangle` control.

Adding a `UIElement` to a Silverlight page is simple enough. Rendering the same element into a `WriteableBitmap` is also simple.

6.4.1 Rendering Silverlight elements

The `WriteableBitmap` class exposes a method named `Render`. The `Render` method draws a Silverlight `UIElement` over the top of the image contained within the `WriteableBitmap`. You already have the stamp declared as a Silverlight `UIElement` (really it's a `Border` control); now you just need to make a call to `Render` from the `Edit_Click` method in MainPage.xaml.cs. The `Edit_Click` method, shown in the following listing, is the event handler for the edit button you added to the application bar.

Listing 6.8 Adding a stamp to the current image

```
private void Edit_Click(object sender, EventArgs e)
{
    if (currentImage != null)
    {
        currentImage.Invalidate();
        var transform = new CompositeTransform
        {
            ScaleX = currentImage.PixelWidth / ContentPanel.ActualWidth,
            ScaleY = currentImage.PixelHeight / ContentPanel.ActualHeight,
            Rotation = -35,
            TranslateX = 100 *
                currentImage.PixelWidth / ContentPanel.ActualWidth,
            TranslateY = 400 *
                currentImage.PixelHeight / ContentPanel.ActualHeight,
        };
        currentImage.Render(photoStamp, transform);
        currentImage.Invalidate();
        imageDetails.Text = "The picture has been stamped.";
    }
}
```

① **Resize photoStamp**

② **Draw stamp**

`WriteableBitmap`'s `Render` method accepts a `UIElement` and a `Transform`. The `UIElement` that you use is the `photoStamp` you just added to MainPage.xaml. The transform is used to position the element within the bitmap. You're going to scale, rotate, and translate the `photoStamp` element using a `CompositeTransform` ①. Even though the image is displayed on the screen at about 480 pixels wide, the bitmap might be much wider. You scale the stamp so that it appears approximately the same size as it does in the `ContentPanel`. You rotate the stamp by 35 degrees. The stamp is translated so that it appears in the lower-right corner of the image. You make a call to the bitmap's `Invalidate` method to ensure the image is properly updated on the screen ②. Finally you update the message displayed to the user.

Try to run the application now, choose your favorite image, and press the Edit button. If all went well, you should see a cool stamp enhancing (or at least obscuring) your favorite picture. The only problem is that you can't save your incredible work of art. You need to wire up the Save button to an event handler so you don't lose any of your hard-earned edits.

6.4.2 Saving an image to isolated storage

The Windows Phone SDK includes a static class named `Extensions`, defined in the `System.Windows.Media.Imaging` namespace. The `Extensions` class provides two extension methods for the `WriteableBitmap` class. The extension methods are named `LoadJpeg` and `SaveJpeg`. The next listing demonstrates how to use `SaveJpeg` to store the picture displayed in the PhotoEditor application to isolated storage.

Listing 6.9 Saving an image to isolated storage

```
using System.Windows.Media.Imaging;

private void Save_Click(object sender, EventArgs e)
{
    if (currentImage != null)
    {
        using (IsolatedStorageFile storage =
            IsolatedStorageFile.GetUserStoreForApplication())
        {
            using (IsolatedStorageFileStream stream =
                storage.CreateFile("customphoto.jpg"))
            {
                currentImage.SaveJpeg(stream, currentImage.PixelWidth,
                    currentImage.PixelHeight, 0, 100);
            }
        }
        imageDetails.Text = string.Format(
            "Image saved with filename:\ncustomphoto.jpg");
    }
}
```

❶ **Include Imaging namespace**

❷ **Write image to file**

You start by adding a using statement for the `System.Windows.Media.Imaging` namespace ❶ where the `SaveJpeg` extension method is found. After checking whether there's a current image, you get the isolated storage container and create a new file named customphoto.jpg. A real application would probably use a filename entered by the user instead of the hard-coded filename used here. You pass the file stream to the `SaveJpeg` method ❷, passing the picture's width and height.

The `SaveJpeg` method declares two other parameters named `orientation` and `quality`. The `orientation` parameter isn't used and should be set to zero. The `quality` parameter allows you to trade image quality for file size. The allowed range of quality values is 0 to 100. Smaller values will result is smaller files, with a trade-off of reduced image quality.

The `SaveJpeg` method makes saving captured or edited pictures really easy. Loading pictures is just as easy.

6.4.3 *Loading an image from isolated storage*

Now that you're saving an image to a file in isolated storage, how do you reload the image back into the application? You already have all the tools you need to load a JPEG file into memory. You'll use `IsolatedStorageFile` to open a file stream, and `PictureDecoder` to read the file stream and load the image into a `WriteableBitmap`. Next you'll add a new menu item to the application bar in MainPage.xaml to allow the user to load an image:

```
<shell:ApplicationBarMenuItem Text="open custom photo"
    Click="Open_Click" />
```

The menu item's `Click` event is handled by a method named `Open_Click` in MainPage .xaml.cs. The following listing shows the implementation of the `Open_Click` method.

Listing 6.10 Loading an image from isolated storage

```
private void Open_Click(object sender, EventArgs e)
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (storage.FileExists("customphoto.jpg"))
        {
            using (IsolatedStorageFileStream stream =
                storage.OpenFile("customphoto.jpg", FileMode.Open))
            {
                currentImage = PictureDecoder.DecodeJpeg(stream);
                photoContainer.Fill =
                    new ImageBrush { ImageSource = currentImage };
            }
            imageDetails.Text = string.Format(
                "Image loaded from filename:\ncustomphoto.jpg");
        }
        else
        {
            photoContainer.Fill = new SolidColorBrush(Colors.Gray);
            imageDetails.Text = "Image not found!";
        }
    }
}
```

❶ **Open file**

❷ **Decode image stream**

❸ **Clear screen**

You start by getting a reference to the isolated storage container and checking whether a file named customphoto.jpg exists. If the file does exist, you open the file ❶ and get a reference to the image stream. The image stream is passed to the `DecodeJpeg` method ❷ and the resulting `WriteableBitmap` is used as the `photoContainer`'s `Fill` brush. If the file doesn't exist, the `photoContainer` is cleared with a solid gray brush ❸.

You're not using the `LoadJpeg` method introduced in the last section for a couple of reasons. `LoadJpeg` requires foreknowledge of the dimensions of the image to be loaded from the image stream. If the `WriteableBitmap` is created with a height and width that don't match the dimensions of the image in the stream, the result is unpredictable. In some cases, the image will load but will be stretched to fit. In other situations, especially when the height of the image in the stream is greater than the height of the `WriteableBitmap`, the `LoadJpeg` method will terminate your application.

Run the application and choose the new menu option to load the image you saved in the last section. Pretty cool! You're done with the PhotoEditor application for now. In the next chapter, you'll update the PhotoEditor to integrate with the Pictures Hub.

6.5 Summary

We started the chapter with two Windows Phone built-in chooser tasks that were left out of the previous discussion on launchers and choosers. The first built-in task is the `PhotoChooserTask`, which allows the application developer to launch the phone's photo selector application from code so that users can select photos from the media library. Another task is the `CameraCaptureTask`, which can be used for opening the phone camera from your application programmatically. We then moved on to show how to use the PhotoCamera API to access the camera directly instead of using the native camera application exposed via the `CameraCaptureTask`. You learned how to use a `VideoBrush` to render the camera's viewfinder in your application and how to listen for the camera button events to trigger the capture of a picture. We showed you how to use the `WriteableBitmap` class to render Silverlight UI elements into the bitmap when you implemented the stamping feature of the sample application. Finally you learned how to read and write files to isolated storage.

The sample application is fairly limited. Real applications wouldn't hard-code filenames, but would allow the user to enter a filename of their own. We're certain that few people are interested in placing a *Windows Phone 7 in Action* stamp on their pictures. If you allowed the user to create custom stamps with their own messages, colors, and symbols, you might have an appealing application.

We're not finished with the PhotoEditor sample application. In the next chapter, you'll learn how to integrate the PhotoEditor into the Pictures Hub. When viewing a picture in the Pictures Hub, the user will be able to launch the PhotoEditor. You'll also update the PhotoEditor to load pictures from the camera roll.

You'll also learn about a method for accessing the microphone using the XNA Framework. You'll build a VoiceRecorder application to create wave files and store them in isolated storage. We'll also show you how to integrate the VoiceRecorder with the Music + Video Hub.

Before moving on, we'd like to mention a second set of camera APIs that are included in the Windows Phone SDK. The `CaptureDevice` API was first introduced with Silverlight 4 for the browser, providing access to the web cameras and microphones attached to a host computer. The same `CaptureDevice` API is implemented in the version of Silverlight shipping on Windows Phone 7.5 devices, allowing developers access to the video camera and microphone built into a Windows Phone device. You can read more about the `CaptureDevice` API in *Silverlight 5 in Action* by Pete Brown, or from `CaptureDevice` API documentation on the MSDN website.

7

Integrating with the Pictures and Music + Videos Hubs

This chapter covers

- Loading and saving pictures in the Media Library
- Photo viewing and sharing
- Recording voice with the microphone
- Listening to the FM radio

Windows Phone contains several built-in applications, including the phone dialer, email, camera, Bing search, the Pictures Hub, and the Music + Videos Hub. In chapters 4 and 6 you learned how to use launchers and choosers and the PhotoCamera class to interact with the built-in applications. Most of these interactions consisted of sample applications launching the built-in applications. In this chapter you're going to learn how to make the Pictures Hub and the Music + Videos Hub launch your application.

The built-in applications use deep link navigation URLs to launch your application and pass information. The Pictures Hub uses a feature known as *App Connect* to display a list of applications in the hub panorama, as well as in various menus displayed throughout the hub. When viewing a picture, the user can choose to share or view the picture with your application. In this chapter you're going to extend the

PhotoEditor sample application from chapter 6 to leverage App Connect for Windows Phone. Along the way you'll learn to use the `MediaLibrary` from the XNA Framework to read and write pictures to the phone's picture albums.

The other built-in application you'll learn to extend is the Music + Videos Hub. The Music + Videos Hub allows the user to play or watch several different types of audio and video. Music and audio might be stored in the phone media library, but they may also be stored in third-party applications, streamed from the internet, or might even be broadcast over the air as an FM radio signal. Applications that play audio report information to the Music + Videos Hub, which displays information to the user in the pages of the hub's panorama control.

In this chapter you'll build a sample application that records voice using the phone's microphone and saves the recording to a wave file in isolated storage. The sample application will report information about the voice recording to the Music + Videos Hub. The user will then be able to play the voice recordings from the Music + Videos Hub. We also show you how to create and use a background agent to play the voice recordings even when the application isn't running in the foreground.

We wrap up the chapter with a discussion about the FM Radio API and how to use the FM radio in your own application. We have a lot to cover so let's get started with updating the PhotoEditor sample application to read and save pictures in the MediaLibrary.

7.1 Working with pictures in the Media Library

We're going to continue working with the PhotoEditor application you created in the last chapter. When you last left the PhotoEditor, the user was able to edit pictures and save them to isolated storage. The Pictures Hub is where you want to save pictures, as this is where the user expects to see all pictures on the device. The `MediaLibrary` class provides the API you need to save pictures to the Pictures Hub.

The `MediaLibrary` class from the XNA framework provides access to songs, playlists, and pictures in the device's media library. The media library plays a huge role in Windows Phone because it's useful when you want to integrate your applications with the phone's built-in applications. When using the `MediaLibrary`, you'll need to reference the `Microsoft.Xna.Framework` assembly.

The `MediaLibrary` has six properties and we'll categorize them into pictures and audio/videos. We're only going to cover picture-related properties in this section.

7.1.1 Exposing Pictures

The two picture-related classes in the `MediaLibrary` namespace are `Picture` and `PictureAlbum`. Each `Picture` class instance provides information about a picture via the `Name`, `Date`, `Height`, `Width`, and `Album` properties. Picture albums are containers for pictures and other picture albums. The collection of pictures in a `PictureAlbum` is accessed through the album's `Pictures` property. The other `PictureAlbums` contained in the album are exposed with the `Albums` property. You can also retrieve the parent `PictureAlbum` via the `Parent` property.

`Pictures` and `PictureAlbums` are exposed by several properties of the `MediaLibrary` class. Table 7.1 lists the picture-related properties of the `MediaLibrary`.

Table 7.1 MediaLibrary properties that expose pictures

Property name	Description
`Pictures`	A single collection of `Picture` objects representing every picture in the Camera Roll, Saved Pictures, and Sample Pictures albums.
`RootPictureAlbum`	The root of the picture album hierarchy. This is the parent `PictureAlbum` of the Saved Pictures, Camera Roll, Sample Pictures, and Favorite Pictures albums.
`SavedPictures`	A collection of `Picture` objects representing every picture in the Saved Pictures album.

Let's talk about the `RootPictureAlbum` property of `MediaLibrary`. `RootPictureAlbum` provides everthing (including photo albums) from the root folder. To better understand the differences between the `Pictures` and `RootPictureAlbum` properties, look at the Phone > Pictures view in the desktop Zune software, shown in figure 7.1. On the phone, the root folder is named Pictures, and this is what `RootPictureAlbum` returns. Usually, there won't be any pictures in the root folder, but it may have two or three default folders.

The Camera Roll folder is the default location where all photos that the user has taken with the phone's camera will be stored automatically. The Sample Pictures folder comes pre-installed with Windows Phone, and the Saved Pictures folder is the default place where a user saves all images from the internet or an application.

> **NOTE** In the Pictures Hub, the user is able to view Facebook or SkyDrive photos, but those aren't stored on the device. Facebook photos can be saved to the Saved Pictures folder on the phone by tapping the Save to Phone link in the application bar while viewing the photo.

The `Pictures` property of the `MediaLibrary` returns the collection of pictures that have been stored in the media library. Due to the integration with Facebook and Windows

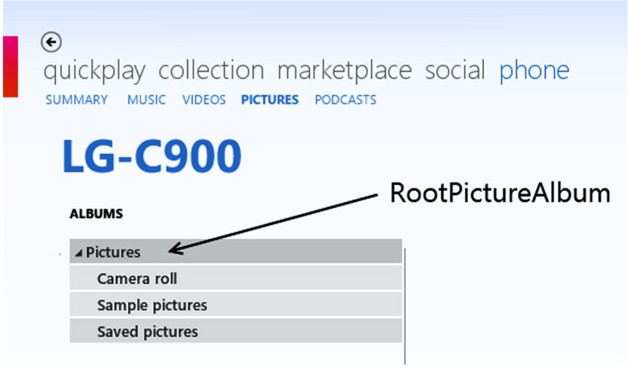

quickplay collection marketplace social phone
SUMMARY MUSIC VIDEOS PICTURES PODCASTS
LG-C900
ALBUMS
RootPictureAlbum
Pictures
Camera roll
Sample pictures
Saved pictures

Figure 7.1 Phone picture albums in the Zune desktop software

Live, more pictures may be visible in the Pictures Hub than will be reported by the `MediaLibrary`. `MediaLibrary` won't provide any information about online photos that haven't been saved to the device yet. You can query all other photos that have been taken with the phone camera, downloaded, or synced from computers.

The last picture-related `MediaLibrary` property is `SavedPictures`. `SavedPictures` returns the collection of pictures stored in the Saved Pictures album. This is a convenience property saving you the hassle of navigating the album hierarchy from `RootPictureAlbum`. The Saved Pictures album is also the only album in the `MediaLibrary` that an application can use when saving pictures.

7.1.2 *Saving pictures to the media library*

In the last chapter you learned how to save a picture to a file in isolated storage. An edited photo seems out of place in isolated storage. All the other pictures on the phone are stored in the Pictures Hub and synchronized with the Zune software on the PC. Now that you have a good understanding of the `MediaLibrary`, we can talk about how to save photos in the Pictures Hub.

Launch Visual Studio and open up the PhotoEditor project. Find the `Save_Click` method in MainPage.xaml.cs and update the code to match the following listing.

Listing 7.1 Saving a picture to the media library

```
using Microsoft.Xna.Framework.Media;

private void Save_Click(object sender, EventArgs e)
{
    if (currentImage != null)
    {
        var stream = new MemoryStream();
        currentImage.SaveJpeg(stream, currentImage.PixelWidth,
            currentImage.PixelHeight, 0, 100);
        stream.Seek(0, 0);

        var library = new MediaLibrary();
        Picture p = library.SavePicture("customphoto.jpg", stream);
        imageDetails.Text = string.Format(
          "Image saved to media library.\r\nFilename:\r\ncustomphoto.jpg");
    }
}
```

❶ **Add using statement**

❷ **Save image to memory stream**

❸ **Save image to SavedPictures album**

Before you can use the `MediaLibrary` class, you declare that you're using the `Microsoft.Xna.Framework.Media` namespace ❶. Next you alter the method to save the image in a `MemoryStream` instead of an `IsolatedStorageFileStream` ❷. Finally you create a `MediaLibrary` instance and call the `SavePicture` method ❸, passing in the memory stream. The `MediaLibrary` doesn't care whether a picture with the name customphoto.jpg already exists. Pictures won't be overwritten and two pictures will be created. When shown in the Zune software on the PC, the duplicate copies will be named customphoto(2).jpg, customphoto(3).jpg, and so on.

Once you've implemented the new `Save_Click` code, you can run the sample project directly on a device from Visual Studio. Use the Choose Picture menu option to select one of the existing pictures in the Pictures Hub. Once the picture is displayed in the PhotoEditor application, click the Edit button followed by the Save button. The application should display the message "Image saved to media library" message if the photo was saved successfully. You can then go the Pictures Hub and check your Saved Pictures album where you'll see the new image.

> **TIP** When working with the media library, you must close the Zune software on the desktop and use the WPconnect.exe tool from the Windows Phone SDK to establish a connection to the phone.

Adding pictures to Pictures Hub is one form of integration with the Pictures Hub and we'll look at other methods later in the chapter. First we look at how to read a picture from the library.

7.1.3 Retrieving a picture from the media library

Retrieving the list of all pictures from the media library is easy. You just need to instantiate a `MediaLibrary` instance and use the `Pictures` property. The `Pictures` property returns a collection of `Picture` objects representing every picture in the Camera Roll, Sample Pictures, and Saved Pictures albums. If you only want a collection from the Saved Pictures album, use the `SavedPictures` property instead.

The `Picture` class not only provides image details through the `Name`, `Date`, `Width`, and `Height` properties, but it also provides access to the stream of bytes that comprise the image contained in the picture file. Actually, the `Picture` class provides access to two image streams—the full size image and a small thumbnail image. These two image streams are accessed via the `GetImage` and `GetThumbnail` methods respectively.

You'll next update the PhotoEditor application to open and edit a picture from the media library. Open MainPage.xaml and add a new menu item to the `ApplicationBar`:

```
<shell:ApplicationBarMenuItem Text="open from library"
    Click="OpenFromLibrary_Click" />
```

The new menu item is wired up to a click event handler named `OpenFromLibrary_Click`. Open MainPage.xaml.cs and add the implementation for the new event handler. The `OpenFromLibrary_Click` implementation is shown in the following listing.

Listing 7.2 Reading a picture from the media library

```
private void OpenFromLibrary_Click(object sender, EventArgs e)
{
    var library = new MediaLibrary();
    var pictures = library.SavedPictures;

    var picture = pictures.FirstOrDefault(
        item => item.Name == "customphoto.jpg");
    if (picture != null)
```

① Find customphoto.jpg

```
    {
        using (var stream = picture.GetImage())
        {
            currentImage = PictureDecoder.DecodeJpeg(stream);
        }
        photoContainer.Fill = new ImageBrush{ ImageSource = currentImage };
        imageDetails.Text = string.Format(
            "Image from Album: {0}\r\nPicture name: {1}",
            picture.Album, picture.Name);
    }
    else
    {
        photoContainer.Fill = new SolidColorBrush(Colors.Gray);
        imageDetails.Text = "Choose an image source from the menu.";
    }
}
```

② **Get image stream**

③ **Clear container**

You start by obtaining a collection of all pictures in the Save Pictures album. You iterate over the collection and find the first picture named customphoto.jpg ❶. Once you find a picture, you use the `PictureDecoder` class to convert the stream returned from `Picture.GetImage` ❷ into a `WriteableBitmap`. You then update the user interface. If a picture isn't found, you fill the user interface with a gray rectangle ❸.

You now know how to load an image from the `MediaLibrary` from within your sample application. In the next section, we show you how to extend the Pictures Hub so that a user can open the picture in your sample application directly from the Pictures Hub.

7.2 Editing and sharing from the Pictures Hub

The Pictures Hub, shown in figure 7.2, is a place where a user can see all of their photos from various sources. All photos that have been taken with the phone, synced from the computer, and downloaded from the internet or email will be included in the Pictures Hub. The Pictures Hub is integrated with Windows Live and Facebook, and all photos uploaded to those websites will be displayed in the Pictures Hub as well.

The apps list is only shown when the user has installed applications that extend the Pictures Hub. Let's see how you can register the PhotoEditor sample application as a Pictures Hub extension.

7.2.1 Extending the Picture Hub

The apps page in the Pictures Hub is extended with a feature known as *App Connect* for Windows Phone. The advantage of App Connect is that users don't need to leave the Pictures Hub in order to use your application. It's already integrated inside the Pictures Hub and the user can use it directly from the Pictures Hub. Applications register with App Connect by including an `Extension` element in the WMAppManifest.xml file that exists in the project's Properties folder:

```
<Extensions>
  <Extension ExtensionName="Photos_Extra_Hub"
    ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5632}"
    TaskID="_default" />
</Extensions>
```

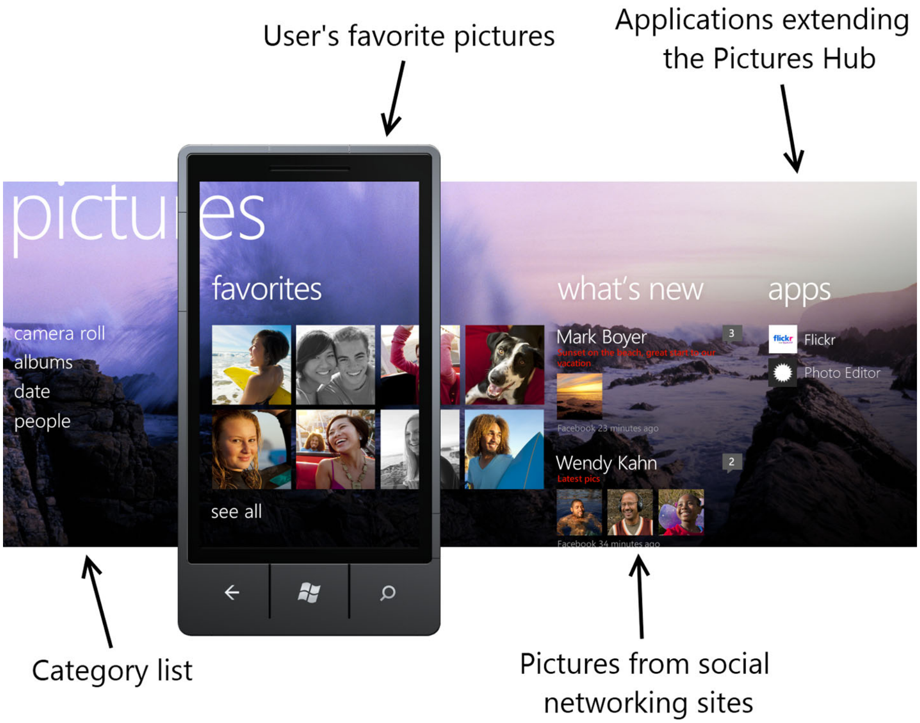

User's favorite pictures
Applications extending the Pictures Hub
pictures
favorites
what's new
apps
camera roll
albums
date
people
see all
Mark Boyer
Facebook 23 minutes ago
Wendy Kahn
Latest pics
Facebook 34 minutes ago
Flickr
Photo Editor
Category list
Pictures from social networking sites

Figure 7.2 Four different sections of the panorama can be seen in this Pictures Hub screen shot. The first section presents a list of categories. The second shows the pictures the user has identified as their favorites. The third section is a running list of new pictures added to the user's social networks. The last section provides a list of applications registered as Pictures Hub extensions.

The `Extension` element is placed inside the `Extensions` element, which resides under the `App` element. The `Extension` element has three attributes named `ExtensionName`, `ConsumerID`, and `TaskID`. The `ConsumerID` field is the unique ID of the application that's being extended and the unique ID representing the Pictures Hub is {5B04B775-356B-4AA0-AAF8-6491FFEA5632}, which is detailed in the App Connect documentation on MSDN. The `TaskID` is the name of the task in your application that should be navigated to when the user taps the application in the Apps page of the Pictures Hub. In Windows Phone 7.1, the only supported `TaskID` is the task named `_default`. We talked about an application's default task in chapter 2.

The `ExtensionName` is defined by the application being extended. The Pictures Hub defines three extensions named `Photos_Extra_Hub`, `Photos_Extra_Viewer`, and `Photos_Extra_Share`. `Photos_Extra_Hub` is the name of the Apps page extension. The other two extensions allow an application to participate in the Apps and Share Menu items available when viewing a single picture in the Pictures Hub.

Open the WMAppManifest.xml file for the PhotoEditor and add a `Photos_Extra_Hub` extension element. Rebuild the PhotoEditor application and deploy it to your

Windows Phone device. Launch the Pictures Hub on your device, and look for the PhotoEditor in the list on the Apps page. Tapping PhotoEditor will launch the application using the normal startup routine.

> **NOTE** The Pictures Hub isn't included in the emulator and a Windows Phone device is required to run a `Photos_Extra_Hub` extension.

Extending the apps list is just the first of three extension points exposed by the Pictures Hub. The second extension point you're going to implement in the PhotoEditor is `Photos_Extra_Viewer`, which allows an application to extend the Picture Viewer.

7.2.2 Extending the Picture Viewer

One of the cool things about Windows Phone is that it allows the developers to register a picture-related application (such as PhotoEditor) as an extension to the Windows Phone built-in picture viewer application. This enables the user who's viewing the photos to use your application directly from the Picture Viewer. Figure 7.3 shows the Picture Viewer displaying an Apps menu item when the user opens an individual photo. If the user clicks on Apps then they'll be taken to the list of photo viewer applications that have been installed on the device.

When the user taps the name in the apps list, the application will be launched and is expected to display the selected picture. Turn the PhotoEditor application into a

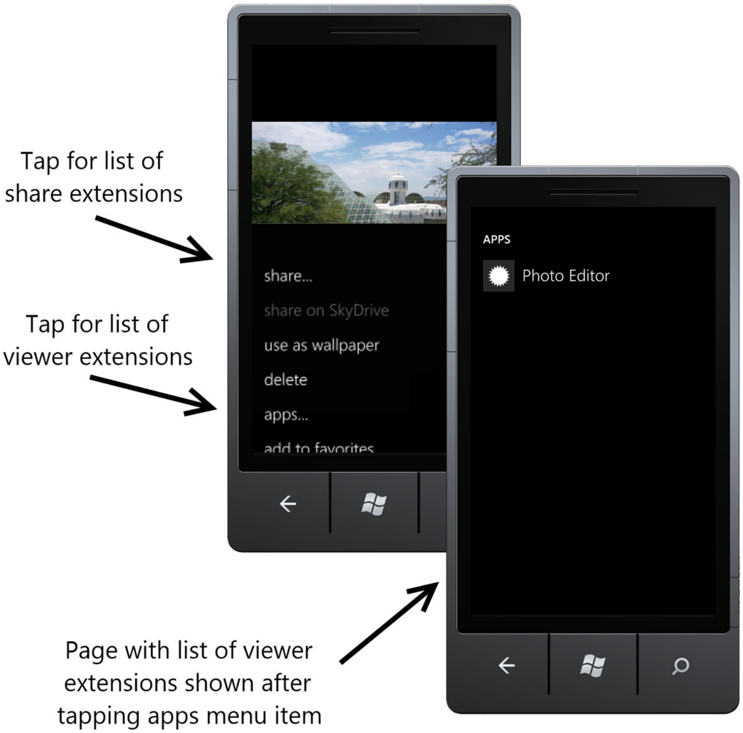

Tap for list of share extensions
Tap for list of viewer extensions
share...
share on SkyDrive
use as wallpaper
delete
apps...
add to favorites
APPS
Photo Editor
Page with list of viewer extensions shown after tapping apps menu item

Figure 7.3 The picture viewer page showing installed Photos_Extra_Viewer applications

picture viewer extension. The first step is to add an `Extension` element with the name `Photos_Extra_Viewer` to the WMAppManifest.xml file:

```
<Extension ExtensionName="Photos_Extra_Viewer"
    ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5632}"
    TaskID="_default" />
```

Once you've rebuilt and deployed the application to a Windows Phone Device, launch Pictures Hub and then choose a photo. Next, expand the application bar's menu and select the apps menu item to see the list of picture viewer applications. You should see your PhotoEditor sample application in the list.

> **TIP** Even though the Photos Hub isn't included in the emulator, you can use a simple trick to run the picture viewer application in the emulator. Once the photo editor application is deployed to the emulator, use the F7 key to mimic a full press of the camera button and the emulator will launch the camera application. Tap on the shutter button in the user interface to take a picture. Finally scroll to the picture you just took and you'll have access to the picture viewer menu.

DETERMINING WHICH PICTURE THE USER IS VIEWING

When the photo viewer launches your application, it passes a token identifying the selected photo. The token is passed as a query string parameter in the navigation URI. The token can then be passed to the XNA `MediaLibrary` to retrieve the corresponding `Picture` object. Let's discuss how to use the token in your sample application, as detailed in the next listing.

Listing 7.3 Using the Pictures Hub token to display the selected photo

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if(State.ContainsKey("customCamera"))
    {
        State.Remove("customCamera");
        InitializeCamera();
    }

    IDictionary<string, string> queryStrings =
       NavigationContext.QueryString;

    string token = null;
    string source = null;
    if (queryStrings.ContainsKey("token"))
    {
        token = queryStrings["token"];
        source = "Photos_Extra_Viewer";
    }

    if (!string.IsNullOrEmpty(token))
    {
        MediaLibrary mediaLib = new MediaLibrary();
        Picture picture = mediaLib.GetPictureFromToken(token);
```

❶ Get query strings sent on URI

❷ Extract token value

❸ Get picture from media library

```
    currentImage = PictureDecoder.DecodeJpeg(picture.GetImage());
    photoContainer.Fill = new ImageBrush{ ImageSource = currentImage };
    imageDetails.Text = string.Format("Image from {0}.\n
➥ Picture name:\n{1}\nMedia library token:\n{2}",
      source, picture.Name, token);
  }
}
```

You're updating the `OnNavigatedTo` method in MainPage.xaml.cs that you created in the last chapter. You get the query strings parameters sent to the application from the `QueryString` property of `NavigationContext` ❶. You check whether a token was passed from Pictures Hub ❷ and if so, you save the value in the `token` variable. You also note that the source of the token is the `Photos_Extra_Viewer` extension. Next, you get the picture from `MediaLibrary` by using the token that you received ❸. Finally, you display the image in the `photoContainer` control and update the message displayed in the `imageDetails` `TextBlock`.

It's not possible to debug an application that's launched by the Pictures Hub. But you can use a simple trick to mimic a Pictures Hub launch and debug your `OnNavigatedTo` code. Open the WMAppManifest.xml file and look for the `DefaultTask` element. This element's `NavigationPage` attribute contains the URL called by the Silverlight Application Host when launching your application. If you add query string parameters to the `NavigationPage` URL, they'll be passed your application when it's launched.

To debug and test the picture viewer application, you first need to know the token value for a picture in the `MediaLibrary`. Fortunately, your sample application displays the token on the screen. Make note of the token and add it to the `NavigationPage` URL:

```
<DefaultTask Name="_default" NavigationPage="MainPage.xaml?
➥ token=%7B60C0B7CD-669A-2DF4-6B1E-CCABD81E168B%7D" />
```

A Picture Viewer extension is typically an application that displays or edits a picture. The last Pictures Hub extension is for applications that share pictures with social networking, messaging, or other web-based applications.

7.2.3 Sharing pictures from your Pictures Hub extension

The Windows Phone Pictures Hub has built-in support for sharing pictures with social networking applications, text messaging, email, and SkyDrive. Third-party developers can register their picture-sharing application as an extension to the Pictures Hub. This enables the user to activate your application from the Share menu that appears from several locations in the Pictures Hub, one of which is shown in figure 7.3. The Share menu activates the Share Picker, which displays all applications registered as `Photos_Extra_Share` extensions. You'll now turn the PhotoEditor application into a Share Picker extension. The first step is to add an `Extension` element with the name `Photos_Extra_Share` to the WMAppManifest.xml file:

```
<Extension ExtensionName="Photos_Extra_Share"
  ConsumerID="{5B04B775-356B-4AA0-AAF8-6491FFEA5632}"
  TaskID="_default" />
```

When the Pictures Hub launches a sharing application, it passes the pictures token to the application. A sharing extension receives the pictures token in the query string parameter named `FileId`. Update the `OnNavigatedTo` method in MainPage.xaml to look for the `FileId` parameter:

```
if (queryStrings.ContainsKey("token"))
{
    token = queryStrings["token"];
    source = "Photos_Extra_Viewer";
}
else if (queryStrings.ContainsKey("FileId"))
{
    token = queryStrings["FileId"];
    source = "Share";
}
```

The code snippet adds an `else` expression to check whether `queryStrings` contains a key named `FileId`. If `FileId` is found, the token variable is set and the source of the token is the `Photos_Extra_Share` extension.

You can use the same `DefaultTask` trick to debug your share extension code, if you know a valid picture token:

```
<DefaultTask Name="_default" NavigationPage="MainPage.xaml?Action=
➥ ShareContent&amp;FileId=%7B60C0B7CD-669A-2DF4-6B1E-CCABD81E168B%7D" />
```

This wraps up the PhotoEditor application that you've been working on for a chapter and a half. The application integrates with the camera and the photo chooser and extends the Picture Hub. But the Picture Hub isn't the only application that provides extension points. The next application you learn how to extend is the Music + Videos Hub.

7.3 Playing and recording with the Music + Videos Hub

You're going to learn about another cool hub called Music + Videos Hub in this section. You'll first learn what the Music + Videos Hub is from an end-user point of view. Then, you'll see how you can develop an application that integrates with the Music + Videos Hub.

The Music + Videos Hub is the central place where you can find all music, videos, and podcast activity on the device. The Music + Videos Hub is launched from its tile on the start screen or from its icon in the Application List. Let's take a look at the architecture of this hub.

The Music + Videos hub is divided into the four sections shown in figure 7.4:

- *Zune* is the central view for playing music, videos, podcasts, radio, and accessing the marketplace.
- *History* contains the list of music, videos, playlists, artists, podcasts, and FM radio stations that the user recently played.
- *New* contains the list of new music, videos, or podcasts that have recently been synced to the phone or downloaded from the marketplace. This list is updated

when media is added to the device or when the user creates an object in a music + videos application.

- *Apps* contains the list of applications installed on the device that integrate with the Music + Videos Hub.

When you're developing for the Music + Videos Hub application, it's crucial to keep these four sections in mind.

To demonstrate how to integrate with the Music + Videos Hub, you need a new sample application. The application will use the phone's microphone to record voice recordings. You might use this feature to create a pronunciation training program that helps users to record their voice and practice until they can pronounce properly. A lot of people who like to learn different languages may find such a program useful.

When a new file is recorded, your application will create an item in the new section of the Music + Videos Hub. The application will also allow the user to play their recordings, which will create items in the Music + Videos history view. Because the sample application integrates with the Music + Videos Hub, it'll be shown in the apps list. When the user taps any of the items you add to the history, new, or apps sections, your application will be launched by the Music + Videos Hub.

You start by creating a new Silverlight Windows Phone Application project named VoiceRecorder. The media and microphone APIs you'll be using are in the XNA Framework so you need to add a reference to the `Microsoft.Xna.Framework` assembly. The application you're going to build is shown in figure 7.5.

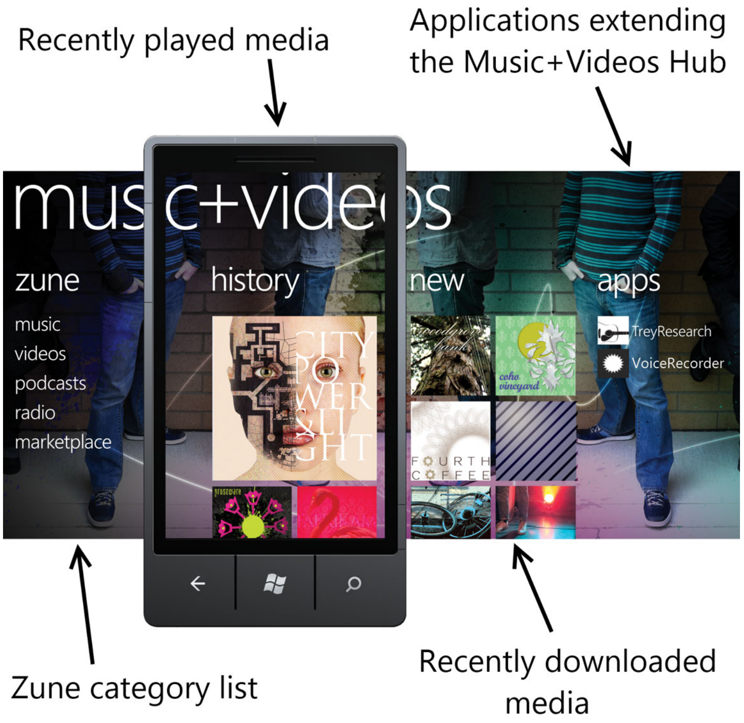
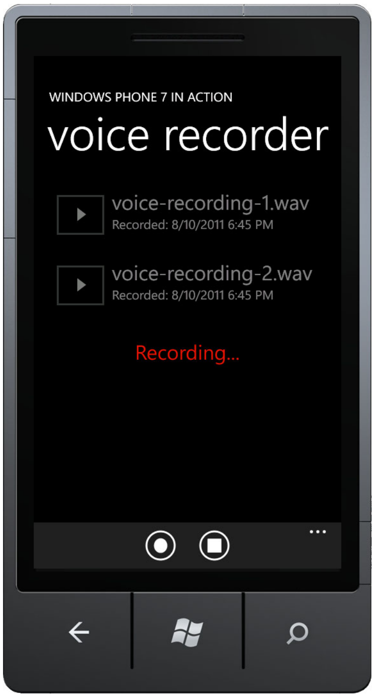

Recently played media
Applications extending the Music+Videos Hub
music+videos
zune
music
videos
podcasts
radio
marketplace
history
new
apps
TreyResearch
VoiceRecorder
Zune category list
Recently downloaded media

Figure 7.4 The four different sections of Music + Videos Hub. The first section is the list of media categories in the Zune. The second section lists the media recently played by the user. Newly downloaded media is shown in the third section. The last section provides a list of applications that extend the Music + Videos Hub.

There is one incompatibility between XNA and Silverlight—the two frameworks utilize different event systems. In order to record voice data, you need to use the XNA event system. That means you need to pump the XNA dispatcher in order to get the microphone events to fire. We'll first take a look how to enable the XNA Framework events in a Silverlight application.



WINDOWS PHONE 7 IN ACTION

voice recorder

voice-recording-1.wav
Recorded: 8/10/2011 6:45 PM

voice-recording-2.wav
Recorded: 8/10/2011 6:45 PM

Recording...

Figure 7.5 The VoiceRecorder sample application

7.3.1 Enabling XNA Framework events

XNA Framework events are dispatched by the `Update` method of the `FrameworkDispatcher` class. In XNA Game projects, the `FrameworkDispatcher` is called automatically by the XNA Framework. The Silverlight framework doesn't automatically call XNA's `FrameworkDispatcher`. Because you're using XNA's `Microphone` class in a Silverlight project, you'll have to call `FrameworkDispatcher.Update` manually to dispatch messages that are in the XNA Framework's message queue.

The `Update` method must be called several times each second to properly process each event. An efficient method for dispatching an XNA event is to place the call to `Update` inside the `Tick` event handler of a `DispatchTimer` object. Add a new `DispatchTimer` field named `xnaTimer` to the `App` class in App.xaml.cs. Initialize the field in the App constructor, wiring up the `Tick` event with a lambda expression:

```
xnaTimer = new DispatcherTimer{ Interval = TimeSpan.FromMilliseconds(16) };
xnaTimer.Tick += (sender, e) => FrameworkDispatcher.Update();
xnaTimer.Start();
```

The lambda expression makes a call to the `Update` method. The `Tick` event is raised every 16 milliseconds, or about 60 times per second. That's all you need to do to enable XNA events in a Silverlight project. With this bit of housekeeping done, you can move on to the user interface.

7.3.2 Building the user interface

Your sample application will have two application bar buttons to start and stop recording. When the user taps on the Record button, the application will start recording from the microphone. The user taps the second button to stop the recording. The sample application will store the voice recording in isolated storage and add the file name to the playlist. The user can tap on the Play button to play the recorded audio file.

The following listing shows what you need to add to the `ContentPanel` in MainPage.xaml.

Listing 7.4 Adding controls to MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <ListBox x:Name="recordingList">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel Orientation="Horizontal">
                    <Button Click="play_Click" Tag="{Binding Title}"
                        Background="Black">
                        <Image Source=
                            "Images/appbar.transport.play.rest.png" />
                    </Button>
                    <TextBlock Text="{Binding Title}"
                        VerticalAlignment="Center"
                        FontSize="{StaticResource PhoneFontSizeLarge}" />
               <TextBlock Text="{Binding Date,
➥  StringFormat='Recorded: \{0:g\}'}"
                        FontSize="{StaticResource PhoneFontSizeNormal}" />
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>

    <TextBlock x:Name="recordingMessage" Text="Recording..."
        Visibility="Collapsed" HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Foreground="{StaticResource PhoneAccentBrush}"
        FontSize="{StaticResource PhoneFontSizeLarge}" />

</Grid>

...

<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton Text="record"
            IconUri="/Images/appbar.record.png"
            Click="record_Click" />
        <shell:ApplicationBarIconButton Text="stop"
            IconUri="/Images/appbar.stop.png"
            Click="stopRecord_Click" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

❶ **Add play button to template**

❷ **Hidden recording message**

❸ **Start and stop recording buttons**

The user interface is built using a `ListBox` to display each voice recording stored in isolated storage. The `ListBox`'s `ItemTemplate` displays the name of each recording and the date and time when the file was recorded. The template also places a `Button` ❶ next to each item in the list. The button will be used to start and pause playback of the recording. You use data binding to store the filename in the buttons `Tag` property. The button displays an image from the Windows Phone SDK image library.

The listing declares a `TextBlock` containing the message "Recording…" that is initially hidden ❷. The `TextBlock` will be shown to the user when the application is actively recording from the microphone. The message will be hidden once recording stops.

The `ApplicationBar` contains two buttons to start and stop a recording ❸. The buttons use custom images, which you can find in the book's sample source code. Make sure you create a project folder named Images, and add the three image files to the project with a build action of `Content`.

Once you've added the required controls in XAML, you need to initialize the microphone. The microphone is represented by the XNA Framework class named `Microphone`. The `Microphone` is a singleton and is accessed through the static `Default` property. Before using the `Microphone` class, you need to set the `BufferDuration` property and wire up the `BufferReady` event in the `MainPage` constructor:

```
public MainPage()
{
    InitializeComponent();

    Microphone.Default.BufferDuration = TimeSpan.FromSeconds(1);
    Microphone.Default.BufferReady += microphone_BufferReady;
}
```

While recording, the `Microphone` stores audio data in an internal buffer. When the buffer is full, the `Microphone` raises the `BufferReady` event. The `BufferDuration` field specifies how much audio will fit into the buffer. Expressed another way, the `BufferDuration` determines the `TimeSpan` between each `BufferReady` event.

There's one last bit of supporting code you need before you can start recording audio with the `Microphone`. You need a simple class to hold the title and recording date. Create a new class named `VoiceRecording` with `Title` and `Date` properties:

```
public class VoiceRecording
{
    public string Title { get; set; }
    public DateTime Date { get; set; }
}
```

With the new class in place you're ready to move on to the main feature of the sample application—recording audio with the `Microphone`.

7.3.3 Recording audio

The `Microphone` raises the `BufferReady` event once after recording audio for the duration specified in the `BufferDuration` property. In the `BufferReady` event handler, audio data should be copied from the `Microphone` into a storage location. In the sample application, this audio data is temporarily copied into a `byte` array and then written to a `MemoryStream`. Add two new fields to the `MainPage` class for the `byte` array and the `MemoryStream`:

```
private MemoryStream audioStream = null;
byte[] audioBuffer = null;
```

The `BufferReady` event is an XNA Framework event, and is raised by the `FrameworkDispatcher`. In the event handler, you get the data from the microphone and write that data to the `MemoryStream`:

```
void microphone_BufferReady(object sender, EventArgs e)
{
    int count = Microphone.Default.GetData(audioBuffer);
    audioStream.Write(audioBuffer, 0, count);
}
```

The `Microphone`’s `GetData` method copies data into the `audioBuffer` `byte` array, and returns the number of bytes written. You use the count of bytes written to the buffer when writing to the `audioStream`.

The `BufferReady` event won’t be triggered until the user starts recording with the microphone. The user starts a recording by tapping the Record button in the user interface. The Record button is wired to the `record_Click` method, which is detailed in the next listing.

Listing 7.5 Recording with the `Microphone`

```
private void record_Click(object sender, EventArgs e)
{
    if (Microphone.Default.State == MicrophoneState.Stopped)
    {
        recordingList.IsEnabled = false;
        recordingMessage.Visibility = Visibility.Visible;

        audioStream = new MemoryStream();
        audioBuffer = new byte[Microphone.Default.
            GetSampleSizeInBytes(TimeSpan.FromSeconds(1))];

        Microphone.Default.Start();
    }
}
```

① **Display recording message**

② **Allocate temporary storage**

The `Microphone` has a property named `State` that you check to determine whether the microphone is already on. If not, then you start the microphone. When the microphone is recording, the `recordingList` `ListBox` is disabled and the recording message in shown to the user ①. The user will be unable to select or play any existing recordings while the current track is being recorded. The temporary `byte` array and `MemoryStream` are allocated ② before you start recording. Recording is started by calling the `Start` method of the singleton `Microphone` instance.

Recording will continue until the user taps the Stop button and triggers the `stopRecord_Click` event handler. The next listing shows the implementation of the `stopRecord_Click` event handler.

Listing 7.6 Stop the recording

```
private void stopRecord_Click(object sender, EventArgs e)
{
    if (Microphone.Default.State == MicrophoneState.Started)
    {
        Microphone.Default.Stop();
        string filename = WriteFile(audioStream);
```

① **Write audio data to file**

```
        audioBuffer = null;
        audioStream = null;

        recordingMessage.Visibility = Visibility.Collapsed;
        recordingList.IsEnabled = true;
        recordingList.Items.Add(new VoiceRecording
            { Title = filename, Date = DateTime.Now });
    }
}
```

② **Add recording to ListBox**

Before you do any work in the `stopRecord_Click` method, you check whether the microphone's state is `Started`. After the microphone has been stopped, you write the contents of the `MemoryStream` to a file in isolated storage ①, and clean up the temporary variables. The recording list is re-enabled and the recording message is hidden once again. Finally, you create a new `VoiceRecording` instance and add it to the items displayed in the `ListBox` ②.

The listing calls through to a method named `WriteFile` to save the audio data in a file. The audio data recorded by the `Microphone` is stored in raw PCM data format. To make the raw PCM data into a real audio file, you must wrap the audio data with a media container. In this application, you're using a wave file container format. A *wave file* is merely a header block followed by the audio data. The following listing shows the implementation of the `WriteFile` method.

Listing 7.7 Saving audio data to isolated storage

```
private string WriteFile()
{
    string filename;
    using (var storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        int index = 1;
        filename = string.Format("voice-recording-{0}.wav", index);
        while (storage.FileExists(filename))
        {
            index++;
            filename = string.Format("voice-recording-{0}.wav", index);
        }

        using (var file = storage.OpenFile(filename, FileMode.CreateNew))
        using (var writer = new BinaryWriter(file))
        {
            writer.Write(new char[4] { 'R', 'I', 'F', 'F' });
            writer.Write((Int32)(36 + audioStream.Length));
            writer.Write(new char[4] { 'W', 'A', 'V', 'E' });
            writer.Write(new char[4] { 'f', 'm', 't', ' ' });
            writer.Write((Int32)16);

            writer.Write((UInt16)1);
            writer.Write((UInt16)1);
            writer.Write((UInt32)16000);
            writer.Write((UInt32)32000);
            writer.Write((UInt16)2);
```

① **Generate unique file name**

② **Write wave header**

```
            writer.Write((UInt16)16);

            writer.Write(new char[4] { 'd', 'a', 't', 'a' });
            writer.Write((Int32)audioStream.Length);
            writer.Write(audioStream.GetBuffer(), 0,
                (int)audioStream.Length);

            writer.Flush();
            file.Flush();
        }
    }
    return filename;
}
```

❸ **Write MemoryStream to the file**

You start by getting a reference to the isolated storage device. You generate a new file name ❶ by appending a number to the end of a hard-coded name and looping until you find a name that's not already used. Next you create and open the file and create a `BinaryWriter`. You use the `BinaryWriter` to write the wave header ❷ and `WaveFormatEx` information. Finally you write the PCM data stored in the `MemoryStream` to the end of the file ❸.

NOTE Details of the wave header and the PCM format are beyond the scope of this book. You can read more about wave files on MSDN at http://mng.bz/6Xe9.

In listing 7.7, you added the newly saved voice recording to the display. You really should be displaying all of the voice recordings saved in isolated storage. You'll now create a new method named `DisplayRecordingNames` to read the names of all the recordings and add them to the `recordingList`. Add a call to the new method at the end of the `MainPage` constructor. The `DisplayRecordingNames` implementation is shown in the next listing.

Listing 7.8 Displaying stored recordings

```
private void DisplayRecordingNames()
{
    using (var storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        var filenames = storage.GetFileNames("*.wav");
        foreach (var filename in filenames)
        {
            var lastWrite = storage.GetLastWriteTime(filename);
            recordingList.Items.Add(new VoiceRecording
                { Title = filename, Date = lastWrite.DateTime });
        }
    }
}
```

❶ **Get all filenames in root folder**

❷ **Display recording information**

After opening the storage device, you get the list of every wav file in the root folder. You loop over the list, retrieving the date and time the file was written, and then add each recording to the `ListBox`.

If you run the VoiceRecorder application now you'll be able to press the Record button and speak into the phone. Your voice will be recorded, and when you press

Stop, the recording will be saved into a wave file in isolated storage. The next feature you'll implement is playing the voice recording.

7.3.4 *Playing audio*

In addition to providing the `Microphone` class to record audio, the XNA Framework provides classes to play audio files. The two classes you're going to use in the VoiceRecorder application are `SoundEffect` and `SoundEffectInstance`. The `SoundEffect` class represents an audio file or stream. The `SoundEffectInstance` wraps the `SoundEffect`, allowing the developer to start, pause, and stop playback of a `SoundEffect`.

The `SoundEffect` class knows how to work with and decode several different types of media containers and audio formats, including wave and PCM. A `SoundEffect` is initialized using a stream containing the audio file. The following listing demonstrates how to open a file stream, initialize a `SoundEffect`, create a `SoundEffectInstance`, and start playback.

Listing 7.9 Playing the recorded voice

```
private SoundEffectInstance audioPlayerInstance = null;

private void PlayFile(string filename)
{
    using (var storage = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (storage.FileExists(filename))
        {
            var fileStream = storage.OpenFile(filename,
                FileMode.Open, FileAccess.Read);

            var soundEffect = SoundEffect.FromStream(fileStream);
            audioPlayerInstance = soundEffect.CreateInstance();
            audioPlayerInstance.Play();
        }
    }
}
```

❶ Declare audio player field

❷ Open audio file

❸ Create SoundEffectInstance

First you add a new field to the `MainPage` class ❶ to hold the currently playing voice recording. Next you create a new method named `PlayFile` that's responsible for opening and playing an audio file. The method opens the file stream in read-only mode ❷. The file stream is used to create a new `SoundEffect` using the static `FromStream` method ❸. The `audioPlayerInstance` field is assigned to the `SoundEffectInstance` class returned from the `CreateInstance` method. Finally, you start playback using the `Play` method of the `SoundEffectInstance`.

The `PlayFile` method accepts a filename from the calling code. In this case, the calling code is the `Click` event handler for the Play button declared in the `ListBox ItemTemplate`. The event hander code is detailed in the next listing.

Listing 7.10 Handling click events for the play button

```
private void play_Click(object sender, EventArgs e)
{
    if (audioPlayerInstance != null &&
        audioPlayerInstance.State == SoundState.Playing)
    {
        audioPlayerInstance.Pause();
    }
    else
    {
        var button = (Button)sender;
        string filename = (string)button.Tag;
        PlayFile(filename);
    }
}
```

When the user clicks on the play button in the `ListBox`, you examine the `State` property to see whether the audio player is currently playing an audio file. If a file is playing, you stop playback by calling the `Pause` method and exit the event handler. If the audio player isn't busy, you get the filename from the `Tag` property of the `Button` control. The `Button` control is sent to the event handler in the `sender` property. You may remember that you bound the `Tag` property to the filename in the `ListBox` `ItemTemplate`. You send the retrieved file name to the `PlayFile` method.

You've now created a simple voice recorder for Windows Phone. Run the application and you should see a list of voice recordings. Tap the Play button next to one of the recordings and you should hear that recording play through the phone's speaker or headphones. Applications that play audio files, especially music tracks, should consider extending the Music + Videos Hub.

7.4 Playing recorded audio in the Music + Videos Hub

Earlier in the chapter you learned that the Pictures Hub enables extensions with `Extension` elements in the WMAppManifest.xml file. The Music + Videos Hub doesn't work with `Extension` elements when figuring out which applications to display in the apps list. How can you make the VoiceRecorder sample application show up in the apps list? The answer is simple.

The App element in the WMAppManifest.xml file includes an attribute named `HubType`. When the `HubType` attribute is set to 1 your application will be shown in the apps list of the Music + Videos Hub.

NOTE Updating the HubType attribute in WMAppManifest.xml is used for development and testing. When an application is submitted to the Application Marketplace, the certification process will automatically determine the HubType and will overwrite WMAppManifest.xml accordingly.

Certain requirements must be met when creating a Music + Videos Hub application, which we'll cover later in the section. Here's an example of an App element:

```
<App xmlns=""
    ProductID="{ dbf1198b-6030-495a-bc4c-0abcc8e2d521}"
    Title="VoiceRecorder" RuntimeType="Silverlight"
    Version="1.0.0.0" Genre="apps.normal"
    Author="VoiceRecorder author"
    Description="Sample description"
    Publisher="VoiceRecorder"
    HubType="1" >
```

When you play any kind of media from a Music + Videos Hub application, you need to integrate with the built-in Music + Videos Hub. That means that your application needs to meet the following requirements:

- *Now Playing*—When you play any media from your application, the media's details should be displayed in the Now Playing list in Music + Videos Hub.
- *History*—Once the media track has finished playing in your application, you need to add the media's details to the History view.
- *New*—When the user adds new media to your application, such as creating a new voice recording or purchasing a new song through the application, that media's details should be added in the New View list in the hub.

Microsoft established the Music + Videos requirements to ensure a consistent user experience for all media applications.

7.4.1 Fulfilling Music + Videos Hub requirements

The Windows Phone class library provides the `MediaHistoryItem` and `MediaHistory` classes that you can use to fulfill these three requirements. The certification guidelines say that any application using the `MediaHistoryItem` and `MediaHistory` classes is considered to be a Music + Videos application. We'll look first at creating a `MediaHistoryItem`, as shown in the next listing.

Listing 7.11 Creating a `MediaHistoryItem`

```
private MediaHistoryItem createMediaHistoryItem(string fullFileName,
    bool smallSize)
{
    string imageName = smallSize ? "artwork173.jpg" : "artwork358.jpg";
    StreamResourceInfo imageInfo = Application.GetResourceStream(
        new Uri(imageName, UriKind.Relative));

    var mediaHistoryItem = new MediaHistoryItem
    {
        ImageStream = imageInfo.Stream,
        Source = "",
        Title = fullFileName
    };
    mediaHistoryItem.PlayerContext.Add("vrec-filename", fullFileName);

    return mediaHistoryItem;
}
```

Load image from content ❶

❷ **Properties must not be null**

Identity data added to context ❸

The Music + Videos Hub requires media artwork when displaying media items. In this example, the artwork is coming from a file named either artwork173.jpg or artwork358.jpg that has been added to the project using a build action of `Content`. You use the `GetResourceStream` API to load the file and to get access to its image stream ❶. The

`MediaHistoryItem` class has three properties that must be assigned a value ❷. The `Source` property isn't used by the Hub, but is still required to have a non-null value. The `Title` property contains the name of the media and is displayed in the Hub with the artwork.

The Windows Phone certification requirements define specific rules about the images used in the `MediaHistoryItem`. The images must be of type JPEG. The tile image must include your application title or logo. The Now Playing image must be 358 pixels x 358 pixels in size. The history and new item images must be 173 pixels x 173 pixels in size.

One other `MediaHistoryItem` property is `PlayerContext` ❸, which is a dictionary of strings that can be used by an application to identify the media item. The entries placed in the `PlayerContext` are returned to the application when the user selects the media item in the Hub.

The `MediaHistory` class is what you use to integrate with the Music + Videos Hub. There's only one instance of the `MediaHistory` class, which is accessed via the `Instance` property. An application updates the now-playing information in the Hub via `MediaHistory`'s `NowPlaying` property. The best time to update the `NowPlaying` property in your sample application is in the `PlayFile` method, right after `SoundEffectInstance.Play` is called:

```
MediaHistory.Instance.NowPlaying = createMediaHistoryItem(filename, false);
```

When using the `NowPlaying` property, the artwork should be a 358 x 358 pixel image. When creating media artwork, you should be careful to keep the overall size of the image file under the value specified by `MediaHistoryItem.MaxImageSize`, which is about 16 KB.

The `NowPlaying` property can also be used by your application to retrieve the information about the last media item it was playing. This might be handy if the user has exited your application and restarts it sometime in the future. You can restart the last item played. `NowPlaying` will only return the last `MediaHistoryItem` played by your application. It won't return media history for items added by other applications.

When your application is finished playing a media item, it should update the recently played list. The sample application isn't implemented to detect when the `SoundEffect` class finished playing, so you're going to cheat and update the recently played list right after you call `SoundEffectInstance.Play`:

```
MediaHistory.Instance.WriteRecentPlay(
    createMediaHistoryItem(filename, true));
```

Using the `WriteRecentPlay` API will cause a new tile to appear in the Music + Videos Hub History view. Your artwork for the recent play tile should be 173 x 173 pixels.

When your application acquires a new media item, it should update the new item list. You acquire new media items when the user has finished a recording, and the new file is written to isolated storage. Update the `WriteFile` method and add a call to `WriteAcquiredItem`:

```
MediaHistory.Instance.WriteAcquiredItem(
    createMediaHistoryItem(fullFileName, true));
```

Using the `WriteAcquiredItem` API will cause a new tile to appear in the Music + Videos Hub New view. Your artwork for the new item tile should be 173 x 173 pixels.

Run the application, create a voice recording, and play it back. Switch to the Music + Videos Hub and you should see a listing in both the new items list and the recently played list. It's expected that when you tap the listing in the hub, the VoiceRecorder application will be launched and the selected recording will be played. Let's take a look at how to determine when your sample application is launched by the Music + Videos Hub.

7.4.2 Launching from the Music + Videos Hub

When the user clicks on the tile representing your media items in the History or New views in the Music + Videos Hub, your application will be launched and the `MediaHistoryItem`'s `PlayerContext` values will be sent to your application as query string parameters. You can read the query string parameters from the `NavigationContext`'s `QueryString` property in the `OnNavigatedTo` override method:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    IDictionary<string, string> queryStrings =
        NavigationContext.QueryString;
    if (queryStrings.ContainsKey("vrec-filename"))
    {
        PlayFile(queryStrings["vrec-filename"]);
    }
}
```

You'll remember that you saved the voice recording file name in the `MediaHistoryItem`'s `PlayerContext` dictionary with the `vrec-filename` key. In `OnNavigatedTo` you check whether the query string contains a parameter named `vrec-filename`. If the parameter is present, you pass the parameter's value to the `PlayFile` method.

You probably realize by now that you can't debug the application when it's launched from the Music + Videos Hub. You can use the same application manifest technique to mimic a Hub launch. In your sample application, you change the `DefaultTask` element to include a `vrec-filename` query string parameter:

```
<DefaultTask Name="_default"
    NavigationPage="MainPage.xaml?vrec-filename=voice-recording-1.wav" />
```

In this section, you've learned how to integrate an application with the Music + Videos Hub, which is the single access point for all media on the Windows Phone. Applications that play or create media should consider using the `MediaHistory` class to report now-playing and history information. When a foreground application plays audio files, playback will stop once the user switches to another application. Windows Phone applications can use background agents to continue playing audio files when the user switches tasks.

7.5 Playing recorded audio with a background agent

Background agents, introduced in chapter 3, enable an application to execute tasks and perform work even when their host application isn't running. The Windows Phone SDK provides two background agents just for playing audio. The background audio agents and their supporting classes are defined in the `Microsoft.Phone.BackgroundAudio` namespace. The `AudioPlayerAgent` allows applications to play local or remote audio files. The `AudioStreamingAgent` can be used to play audio streamed to the device. In this section you add an `AudioPlayerAgent` to the VoiceRecorder application to enable background playing of the recorded audio.

`AudioPlayerAgents` are created using the Windows Phone Audio Playback Agent project template. Add a new audio playback project to the Visual Studio solution containing the VoiceRecorder project and name the new project VoiceRecorderPlaybackAgent. Using the Add Reference dialog, add a project reference to the VoiceRecorderPlaybackAgent project from the VoiceRecorder project. In addition to adding the project reference, Visual Studio updated the WMAppManifest.xml file so that it contains a new `ExtendedTask` Element:

```
<ExtendedTask Name="BackgroundTask">
    <BackgroundServiceAgent Specifier="AudioPlayerAgent"
        Name="VoiceRecorderPlaybackAgent"
            Source="VoiceRecorderPlaybackAgent"
            Type="VoiceRecorderPlaybackAgent.AudioPlayer" />
</ExtendedTask>
```

The project template created one class named `AudioPlayer` in the VoiceRecorderPlaybackAgent project. Derived from `AudioPlayerAgent`, the `AudioPlayer` class is generated with four overrides named `OnPlayStateChanged`, `OnUserAction`, `OnError`, and `OnCancel`. Before we discuss each of these overrides, let's discuss how the foreground application communicates with the background agent.

Applications communicate with the background agent through the `BackgroundAudioPlayer` class. `Playback` is controlled using the `Play`, `Pause`, and `Stop` methods. The audio player also provides `FastForward`, `Rewind`, `SkipNext`, and `SkipPrevious` methods. Playback progress can be read with `PlayerState`, `Position` and `BufferProgress` properties.

Now that you have a background audio agent, you need to update the VoiceRecorder to use the `BackgroundAudioPlayer` instead of a `SoundEffectInstance` to play and pause audio files. The application tells the `BackgroundAudioPlayer` which audio file to play using the `Track` property. Open MainPage.xaml.cs, find the `PlayFile` method, and replace the code using the `SoundEffect` class and the `audioPlayerInstance` field with the code in the following snippet:

```
Uri fileUri = new Uri(filename, UriKind.Relative);
BackgroundAudioPlayer.Instance.Track = new AudioTrack(
    fileUri, filename, "Windows Phone 7 in Action", null, null, null,
    EnabledPlayerControls.Pause);

BackgroundAudioPlayer.Instance.Play();
```

Once the track is specified, you ask the background agent to play the voice recording by calling the `BackgroundAudioPlayer`'s `Play` method. You access the singleton instance of the `BackgroundAudioPlayer` using the static `Instance` property.

The `Track` property is of type `AudioTrack` and you construct a new `AudioTrack` by specifying the URI to the voice recording in isolated storage, the title, and artist. You use the filename as the track's title, and the string "Windows Phone 7 in Action" as the track's artist. You also specify that only the pause control should be enabled in the Universal Volume Control.

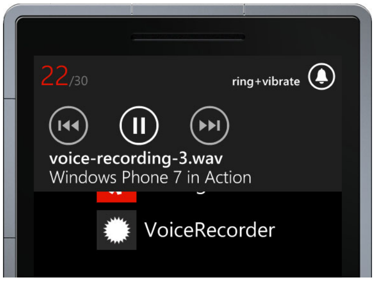


Figure 7.6 Voice recordings in the Universal Volume Control

The host application isn't the only process that controls background audio playback. Background audio agents automatically integrate with the Universal Volume Control (UVC). Normally, the UVC allows the user to fast forward, rewind, skip previous, and skip back. When you created the `AudioTrack`, you specified that only the pause control should be enabled, and the UVC for the VoiceRecorder (shown in figure 7.6) disables the rewind/skip previous and fast forward/skip next buttons.

Remember that the user can pause the voice recording within the VoiceRecorder application by pressing the Play button during playback. You need to replace the pause code in the `play_Click` method so that it also uses the `BackgroundAudioPlayer` instead of the `audioPlayerInstance` field:

```
private void play_Click(object sender, EventArgs e)
{
    if (BackgroundAudioPlayer.Instance.PlayerState == PlayState.Playing)
    {
        BackgroundAudioPlayer.Instance.Pause();
    }

    ...
}
```

The `BackgroundAudioPlayer` reports the current playback state through the `PlayerState` property and the `PlayState` enumeration. The `PlayState` enumeration defines a dozen different states including `Playing`, `Paused`, and `Stopped`. In this code, if the state is `Playing`, you call the `Pause` method.

Note that calling the `Play` or `Pause` methods from the foreground applications doesn't result in the voice recording actually playing. When the foreground application calls `Play`, `Pause`, or the other playback control methods, a message is sent to the background agent. It's the responsibility of the background agent to handle the foreground requests. The `BackgroundAudioPlayer` delivers messages to the background agent with the `OnUserAction` override method.

The `OnUserAction` method is sent information about the action to be performed through a `UserAction` enumeration value. Possible actions include `Stop`, `Pause`, `Play`, `SkipNext`, `SkipPrevious`, `FastForward`, `Rewind`, and `Seek`. The VoiceRecorder sample application only uses `Play` and `Pause`, so those are the only actions your background agent will support.

The `AudioPlayer` class generated by the project template contains a default implementation of the `OnUserAction`. Open the AudioPlayer.cs file and review the generated code. When the user action is `Play` or `Pause`, the corresponding method on the passed-in `BackgroundAudioPlayer` instance is invoked. Calling the `Play` methods from the background application does cause the audio file playback to start or continue. You learned in chapter 3 that `NotifyComplete` method is called by a background agent to inform the operating system that it has successfully completed its work.

Use the `OnPlayStateChanged` override method to detect when the audio track has finished or stopped. `OnPlayStateChanged` will be called with `Stopped` and then with `Shutdown` when the user picks a song from the music library. `OnPlayStateChanged` is called with `TrackEnded` once the audio track finishes.

You've now created a simple voice recorder for Windows Phone. There are a lot of ways that you can improve this application to have a more professional voice recorder. As this is just an example, we'll leave those improvements for you. One other audio media format can be used with the Windows Phone—FM radio.

7.6 Listening to FM radio

In this section, you're going to learn about the built-in FM radio. Microsoft's Windows Phone hardware specification requires all devices to have an FM radio. The radio isn't present in the emulator, so you'll need a device and headphones to test the sample. Please ensure that you close the Zune software on the PC and run WPConnect.exe when you're testing. We're going to walk you through building a sample application in this section, but a working FM radio sample project is included in the book's sample source code.

The Windows Phone class library provides access to the radio hardware with the `FMRadio` class in the `Microsoft.Devices.Radio` namespace. The FM radio is controlled through a singleton instance of the `FMRadio` class. The singleton is exposed to your code through the static `Instance` property. The other properties of the `FMRadio` class are described in table 7.2.

Table 7.2 FMRadio properties

Property name	Description
`CurrentRegion`	A `RadioRegion` enumeration value representing Europe, Japan, or the United States.
`Frequency`	The `Frequency` must be set to a double value appropriate to the current region.
`PowerMode`	The current power state of the radio. The radio is turned on or off by setting the `PowerMode` property.
`SignalStrength`	The received radio strength indicator for the current frequency. The strength will be set to zero if the Universal Volume Control is used to pause radio play.

Windows Phone currently supports three frequency regions—Europe, Japan, and the United States. These aren't true geographic regions, but define a range of frequencies commonly used in these geographic regions. Your country may not be in Europe, Japan, or the United States but would fall into one of the three frequency regions. The appropriate frequency region value depends on how FM stations are broadcasting in your area. For example, in Singapore, there's a popular FM radio station that broadcasts at 91.3 MHz. Phone users in Singapore can listen to the FM radio with Windows Phone even though Singapore isn't located in Europe, Japan, or the United States. You need to set the current frequency region based on the range of frequencies that you want to allow. Otherwise, the `Frequency` property will throw an exception. Table 7.3 shows the regions and their corresponding frequencies.

Table 7.3 FM Radio regions and frequencies

Region	Frequency
US	87.9-107.9 MHz in increment of 0.2
Japan	76.0-90.0 MHz in increment of 0.1
Europe	87.5 to 108.0 MHz

The FM radio hardware is activated and powered on with the `PowerMode` property. An application turns on the radio by setting the `PowerMode` property to `On`, as defined by the `RadioPowerMode` enumeration:

```
FMRadio.Instance.PowerMode = RadioPowerMode.On
```

The FM radio remains powered on when your application is deactivated or exited. Since this can be a potential battery drain, applications should either explicitly turn off the FM radio in the `Deactivated` or `Closing` event handlers, or provide the user with an application setting to leave the FM radio on. If an application leaves the FM radio running, the user can change the FM radio settings with the radio user interface in the Music + Videos Hub.

The user can also use the Universal Volume Control to stop the radio. The user may even change the radio settings with the UVC while the application is running. When the user presses the pause button in the UVC, the radio remains powered on, but the sound is muted and the `SignalStrength` property reports a value of zero.

7.7 Summary

You started the chapter by updating the PhotoEditor sample application introduced in chapter 6. You learned about the Pictures and Music + Videos Hubs. You now understand that those hubs are the applications where all photos, songs, videos, and podcasts are stored in your phone. We analyzed the Pictures Hub from both the end-user and developer perspectives. You registered the PhotoEditor application as an extension of Pictures Hub with App Connect for Windows Phone.

We explored recording the voice using the Windows Phone microphone and extending the Music + Videos Hub. The last section discussed how to access the radio hardware built into the phone. The API for FM radio is simple but you need to keep in mind radio regions and frequencies.

In the next chapter you'll learn how to use the phone sensors, such as the accelerometer and the gyroscope. You'll also look at how to interact with location services.

8

Using sensors

This chapter covers

- Sensor API design
- Interpreting sensor data
- Using sensors in the emulator
- Moving with the motion sensor

Some really cool applications can be built combining sensors with other features of the phone. Applications might respond to a user shaking the device by randomly selecting an object or clearing the screen. Games can use the device's movement as an input mechanism, basically turning the whole phone into a game controller. Another class of applications augments the real world with computer-generated information. Augmented reality apps can show you the location of friends nearby relative to your current location. Astronomy applications determine the position of your device and identify the stars in the night sky. A tourist application might be able to identify nearby landmarks.

All of these applications require sensor input from the physical world. The phone's accelerometer, compass, and gyroscope sensors capture input from the real world and serve the data to applications through the Windows Phone SDK's Sensor API. When combined with location data from the phone's Location Service, stunning augmented reality applications are possible. We discuss the location service in chapter 13.

Dealing with raw data from the sensors can be tricky, especially when trying to calculate the direction in which a device is pointed. The `Motion` sensor takes input from each of the other sensors, performs several complex calculations, and provides data related to motion and a device's relative position in the real world.

In this chapter you're going to build two similar sample applications. The first sample application uses the `Accelerometer`, `Compass`, and `Gyroscope` to demonstrate how the sensors are similar, and how they differ. The second sample application demonstrates how the `Motion` sensor is a wrapper around the three other sensors.

Before we dive into the first sample application, we introduce the common Sensor API that's the foundation for the sensors exposed by the Windows Phone SDK.

8.1 Understanding the sensor APIs

Whereas the `Accelerometer`, `Compass`, `Gyroscope`, and `Motion` sensors each return different types of data, they each implement the same pattern for reporting their data. Over the next several pages, you'll learn techniques that are useful for reading data from any of the sensors. We show you these techniques as you build the foundation of the sample application.

The classes and interfaces that comprise the Sensor API are found in the `Microsoft.System.Devices` namespace which is implemented in the assembly with the same name. Projects that use the Sensor API must include a reference to the `Microsoft.System.Devices` assembly, as well as the `Microsoft.Xna.Framework` assembly. The Sensor API uses the `Vector3`, `Matrix`, and `Quaternion` structs defined in the `Microsoft.Xna.Framework` namespace.

The `Accelerometer`, `Compass`, `Gyroscope`, and `Motion` sensors share a common base class named `SensorBase`. The `SensorBase` class is a generic class where the type parameter is a class that implements `ISensorReading`. We can look at how the `Accelerometer` class is declared in an example:

```
public sealed class Accelerometer : SensorBase<AccelerometerReading>
```

The `ISensorReading` interface is a marker interface defining a single `DateTimeOffset` property named `Timestamp`. The `SensorBase` class defines the common behavior of the sensor implementations. `SensorBase` declares one event and several properties and methods, which are described in table 8.1.

Table 8.1 SensorBase members

Member	Type	Description
`CurrentValue`	Property	The read-only `ISensorReading` containing the currently available sensor data.
`CurrentValueChanged`	Event	The event raised when the `CurrentValue` property changes.

Table 8.1 SensorBase members *(continued)*

Member	Type	Description
`Dispose`	Method	Releases the hardware and other resources used by the sensor.
`IsDataValid`	Property	A read-only property that returns `true` if `CurrentValue` contains valid data.
`Start`	Method	Enables the sensor and begins data collection.
`Stop`	Method	Disables the sensors and ends data collection.
`TimeBetweenUpdates`	Property	Specifies how often the sensor reads new data. The `CurrentValue` property will only change once every time interval.

An application obtains the current sensor reading by calling the `CurrentValue` method. Alternatively, an application can subscribe to the `CurrentValueChanged` event to receive a sensor reading only when the `CurrentValue` property changes. The `CurrentValue` can be read even when the sensor isn't started, but the value returned may not be valid and `IsDataValid` will return false.

> **NOTE** If the `ID_CAP_SENSORS` capability isn't present in the WMAppManifest .xml file, attempts to read the current value, set the time between updates, or start the sensor will result in a `UnauthorizedAccessException`.

Each of the sensor classes defines a static method named `IsSupported`. The `IsSupported` method allows a developer to determine whether the sensor hardware is installed on a particular device and whether the sensor is available to the application. If the `IsSupported` method returns false, attempts to read the current value, set the time between updates, or start the sensor will result in an `InvalidOperationException`.

The Sensor API handles fast application switching on its own. Developers don't need to unhook the sensors when the application is switched from the foreground. Unlike the camera, sensors automatically resume and don't provide an explicit restore method. When the application is resumed, the sensors and events are reconnected and data starts to flow again. Before you learn how to work with the data flowing from the sensors, you need to understand how the sensors report data in three dimensions.

8.1.1 Data in three dimensions

Each of the sensors reports data relative to the x, y, z coordinate system defined by the Windows Phone device. The device's coordinate system is fixed to the device, and moves as the phone moves. The x axis extends out the sides of the device, with positive x pointing to the right side of the device, and negative x pointing to the left side of the device. The y axis runs through the top and bottom of the device, with positive y pointing toward the top. The z axis runs from back to front, with positive z pointing

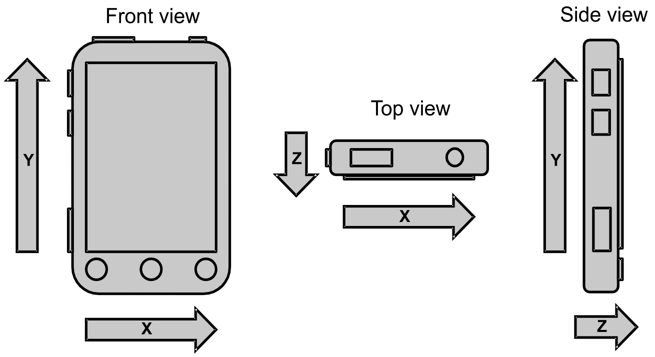


Figure 8.1 The x, y, z coordinate system as defined by a Windows Phone

out the front of the device. Figure 8.1 shows the x, y, and z axes from three different views of a phone.

The coordinate system used by the sensors doesn't necessarily match the coordinate system used by other APIs. One example is the coordinate system used by Silverlight. In portrait mode Silverlight, the y axis points in the opposite direction, with positive y pointing out the bottom of the device. Now that you understand the coordinate system used by the sensors, let's take a closer look at reading data from the sensors.

8.1.2 Reading data with events

Each of the sensors supports an event-driven interaction model with the `CurrentValueChanged` event. The `CurrentValueChanged` event sends a `SensorReadingEventArgs` instance to an event handler. `SensorReadingEventArgs` is a generic class where the type parameter is a class that implements `ISensorReading`. The type parameter matches the type parameter defined by the `SensorBase` class. The `SensorReadingEventArgs` class has a single property name `SensorReading`, which returns the data contained in the sensor's `CurrentValue` property at the time the event was raised.

The `CurrentValueChanged` event handler is called on a background thread. If the event handler updates the user interface, the update logic must be dispatched to the UI thread. The following code snippet shows an example that handles the `CurrentValueChanged` event from the motion sensor:

```
void sensor_CurrentValueChanged(object sender,
    SensorReadingEventArgs<MotionReading> e)
{
    MotionReading reading = e.SensorReading;
    Dispatcher.BeginInvoke(() =>
    {
        // add logic here to update the UI with data from the reading
        ...
    }
}
```

You'll use `CurrentValueChanged` later in the chapter when you build the second sample application. The first sample application will poll for data using the `CurrentValue` property.

8.1.3 Polling for data

An application doesn't need to wait for the sensor to raise an event to ask for data. Each sensor exposes data through the `CurrentValue` property. The `CurrentValue` property can be read whenever the application data determines it needs new data. For example, the reading might be initiated from a button click, a timer tick event, or a background worker:

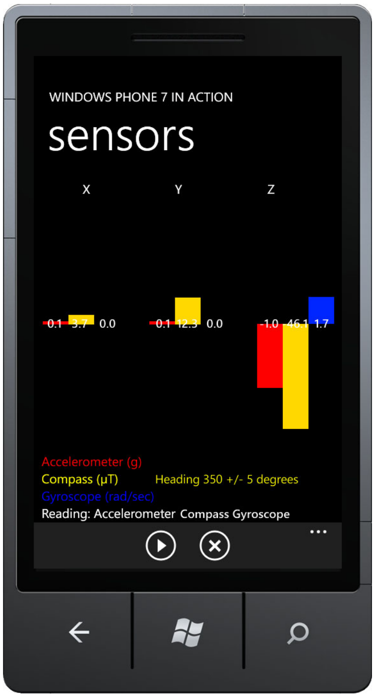
```
if (Compass.IsSupported && compassSensor.IsDataValid)
{
    CompassReading reading = compassSensor.CurrentValue;
    // add logic here to use the data from the reading
    ...
}
```

You'll read sensor data from a timer tick event in your first sample application. Before we can show you the sensors in action, you need to create a new project and prepare the application to display sensor data.

8.2 Creating the sample application

Open Visual Studio and create a new Windows Phone Application named Sensors. The sample application will read values from the `Accelerometer`, the `Compass`, and the `Gyroscope`. By default, Windows Phone Application projects don't reference either the Sensors or the XNA Framework assemblies, and you need to add references to Microsoft.Devices.Sensors.dll and Microsoft.Xna.Framework.dll.

The sample application, shown in figure 8.2, displays a set of colored bars for each of the sensors. Each set of bars displays sensor readings for the x, y, and z coordinates. At the bottom of the screen, the application displays a legend and informational messages about the sensors.


WINDOWS PHONE 7 IN ACTION

sensors

Accelerometer (g)

Compass (µT) Heading 350 +/- 5 degrees

Gyroscope (rad/sec)

Reading: Accelerometer Compass Gyroscope

Figure 8.2 The Sensors sample application

When a sensor's value is positive, a bar will be drawn to scale above the effective zero line. A negative sensor value results in a bar drawn below the zero line. Since the range of possible values differs between each sensor, the height of the bar is transformed from the sensors value into a pixel height using a scaling factor. We'll talk more about each sensor's range of values throughout the chapter. First, you'll create a reusable control to display the positive and negative bars.

8.2.1 Creating a reusable Bar control

To simplify your application, you'll use a reusable control that allows you to set a scale factor and a sensor value. When the scale or value properties change, the control should draw the appropriate positive or negative bar, and display the value with a label. You'll implement the control using the Windows Phone User Control item template. Name the new item `Bar`. The XAML markup for the new control is shown in the next listing.

Listing 8.1 Markup for the Bar control

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
        <RowDefinition Height="1*" />
        <RowDefinition Height="1*" />
    </Grid.RowDefinitions>
    <Rectangle x:Name="positiveBar" VerticalAlignment="Bottom" />
    <Rectangle x:Name="negativeBar" Grid.Row="1" VerticalAlignment="Top" />
    <TextBlock x:Name="label" VerticalAlignment="Center"
        Grid.RowSpan="2" Text="0" TextAlignment="Center" />
</Grid>
```

① **Divide control into two rows**

② **Center label**

The grid is divided into two halves ① with each half containing a `Rectangle`. The first `Rectangle` displays positive values and the other represents negative values. A label is placed right in the middle ② to show the bar's value.

Pages that host a `Bar` control need the ability to set different fill colors for the rectangles. Add a new property named `BarFill` to Bar.xaml.cs code behind file:

```
public Brush BarFill
{
    get { return positiveBar.Fill; }
    set
    {
        positiveBar.Fill = value;
        negativeBar.Fill = value;
    }
}
```

The setter for the `BarFill` property assigns the specified `Brush` to both the `positiveBar` and `negativeBar` rectangles.

> **NOTE** If you were building a proper reusable Silverlight control, the `BarFill` property and the other properties you're going to create would be dependency properties. You'd declare template parts and your XAML markup would be the default template. See the book *Silverlight 5 in Action* for more details on building reusable Silverlight controls.

Next you create properties to set the scale and value for the bar. Since you don't know the full range of values, you need the caller to tell the control how to scale the value to the height of the rectangles. Let's say you need the bar to display a value between 2 and -2, and the `Bar` control is 200 pixels high. A value of 2 would require the positive bar to be a hundred pixels high, whereas a value of -1 would require the negative bar to be 50 pixels high. The next listing details how the bar height is calculated using the `Scale` and `Value` properties.

Listing 8.2 Calculating bar height with the `Scale` and `Value` properties

```
private int scale;
public int Scale
{
    get { return scale; }
    set
    {
        scale = value;
        Update();                        ◁─┐
    }                                      ❶ Recalculate when
}                                            properties change

private float barValue;
public float Value
{
    get { return barValue; }
    set
    {
        barValue = value;
        Update();
    }
}

private void Update()                             ❷ Calculate
{                                                   height of bar
    int height = (int)(barValue * scale);       ◁──┘
    positiveBar.Height = height > 0 ? height : 0;
    negativeBar.Height = height < 0 ? height * -1 : 0;   ◁─┐
    label.Text = barValue.ToString("0.0");                ❸ Invert negative
}                                                           height
```

You implement both the `Scale` and the `Value` properties with backing fields and simple getters and setters. Inside the setter of each property, you call the `Update` method ❶ to recalculate the height of the bar rectangles and update the user interface. Inside the `Update` method you multiply the `scale` and `barValue` fields ❷, and the resulting value is the number of pixels high the bar should be drawn. If the calculated height value is greater than 0, the `positiveBar`’s `Height` is updated to the new value. If the calculated height value is less than zero, you invert the calculated value ❸ before assigning the `negativeBar`’s height. Finally, you use the `ToString` method with a formatting string to set the label’s `Text` property.

Now that you have a bar control, you can create your application’s user interface. You need to add an XML namespace to MainPage.xaml so that you can use your new bar control:

```
xmlns:l="clr-namespace:Sensors"
```

You’re now ready to use the `Bar` control in the MainPage’s XAML markup. You need to design the MainPage to have three `Bar` controls for each sensor, for a total of nine `Bar` controls.

8.2.2 Designing the main page

If you look at the screenshot in figure 8.2, you'll notice that MainPage.xaml is divided into three rows and several columns. The markup for the `ContentPanel` of MainPage.xaml is shown in the next listing.

Listing 8.3 Markup for MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Grid.RowDefinitions>
    <RowDefinition Height="25" />
    <RowDefinition Height="400" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
  </Grid.ColumnDefinitions>

  <TextBlock Text="X" Grid.Column="1" />
  <TextBlock Text="Y" Grid.Column="5" />
  <TextBlock Text="Z" Grid.Column="9" />

  <l:Bar x:Name="accelX" Grid.Row="1" Grid.Column="0"
    BarFill="Red" Scale="100" />
  <l:Bar x:Name="accelY" Grid.Row="1" Grid.Column="4"
    BarFill="Red" Scale="100" />
  <l:Bar x:Name="accelZ" Grid.Row="1" Grid.Column="8"
    BarFill="Red" Scale="100" />

  <l:Bar x:Name="compassX" Grid.Row="1" Grid.Column="1"
    BarFill="Yellow" Scale="4" />
  <l:Bar x:Name="compassY" Grid.Row="1" Grid.Column="5"
    BarFill="Yellow" Scale="4" />
  <l:Bar x:Name="compassZ" Grid.Row="1" Grid.Column="9"
    BarFill="Yellow" Scale="4" />

  <l:Bar x:Name="gyroX" Grid.Row="1" Grid.Column="2"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroY" Grid.Row="1" Grid.Column="6"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroZ" Grid.Row="1" Grid.Column="10"
    BarFill="Blue" Scale="32" />

  <StackPanel Grid.Row="2" Grid.ColumnSpan="11">
    <TextBlock Foreground="Red" Text="Accelerometer (g)" />
    <TextBlock x:Name="heading" Foreground="Yellow" Text="Compass (uT)"/>
```

① **Eleven columns**

② **Three bars for each sensor**

③ **Legend and messages**

```
    <TextBlock Foreground="Blue" Text="Gyroscope (rad/sec)" />
    <TextBlock x:Name="messageBlock" Text="Press Start" />
  </StackPanel >
</Grid>
```

You start by dividing the `ContentPanel` into three rows and eleven columns ❶. The first row contains three `TextBlocks` serving as the titles for the x, y, and z coordinates. The second row shows three bars ❷ for each of the `Accelerometer`, `Compass`, and `Gyroscope` sensors. The `Bar` controls are 400 pixels high divided into positive and negative sections of 200 pixels each. Allowing for three columns for each sensor, and two spacer columns, you need a total of eleven columns. The last row ❸ contains a legend and messages.

Each `Bar` control is assigned a `BarFill` color—red for accelerometer values, yellow for compass values, and blue for gyroscope values. Each `Bar` control is also assigned a scale value. We'll describe how the scale factors were calculated in our detailed discussion of each sensor later in the chapter.

The last things you need for your application are application bar buttons to start and stop the sensors:

```
<phone:PhoneApplicationPage.ApplicationBar>
  <shell:ApplicationBar IsVisible="True" IsMenuEnabled="False">
    <shell:ApplicationBarIconButton Text="start" Click="start_Click"
      IconUri="/Images/appbar.transport.play.rest.png" />
    <shell:ApplicationBarIconButton Text="stop" Click="stop_Click"
      IconUri="/Images/appbar.cancel.rest.png" />
  </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Create a project folder named Images and add the two images to the project. The images can be found in the Icons folder of the Windows Phone 7.1 SDK. Be sure to set the image's build property to `Content`.

8.2.3 Polling sensor data with a timer

In the sample application you update the screen with data from three different sensors. To simplify the logic, you're not going to use the `CurrentValueChanged` events for the sensors, and will use a polling method instead. You'll use a `DispatchTimer` to update the user interface about 15 times a second. Add a `DispatchTimer` field:

```
DispatcherTimer timer;
```

You initialize the timer field inside the `MainPage` constructor. You ask the timer to `Tick` every 66 milliseconds or about 15 times a second. You'll poll each of the sensors inside the `timer_Tick` method:

```
public MainPage()
{
    InitializeComponent();

    timer = new DispatcherTimer();
    timer.Tick += timer_Tick;
    timer.Interval = TimeSpan.FromMilliseconds(66);
}
```

The timer is started in the `start_Click` method. When the timer is started, you update the message displayed in the `TextBlock` named `messageBlock` to let the user know which sensors have been started:

```
private void start_Click(object sender, EventArgs e)
{
    if (!timer.IsEnabled)
    {
        string runningMessage = "Reading: ";
        timer.Start();
        messageBlock.Text = runningMessage;
    }
}
```

You'll add additional code to the `start_Click` method as you hook up the `Accelerometer`, `Compass`, and `Gyroscope` later in the chapter.

The timer is stopped in the `stop_Click` method. When the timer is stopped, you update the message displayed in the user interface:

```
private void stop_Click(object sender, EventArgs e)
{
    timer.Stop();
    messageBlock.Text = "Press start";
}
```

You'll also update the `stop_Click` method as you hook up the sensors in later sections of the chapter.

You could run the application now to check that the form is laid out as you expect, but the application doesn't do anything interesting yet. You'll remedy that by adding in the accelerometer sensor.

8.3 Measuring acceleration with the accelerometer

The accelerometer can be used in games and applications that use phone movement as an input mechanism or for controlling game play. The Accelerometer tells you when the device is being moved. It can also tell you whether the device is being held flat, at an angle, or straight up and down.

The accelerometer measures the acceleration component of the forces being applied to a device. Note that acceleration due to gravity isn't reported by the sensor. Unless the device in is free fall, forces are always being applied to a device. The `Accelerometer` reports numbers in terms of the constant g, which is defined as the acceleration due to Earth's gravity at sea level. The value of g is -9.8 m/s^2.

When a device is at rest lying on a table, the table is exerting a force on the device that offsets the pull of gravity. The `Accelerometer` measures the acceleration of the force the table applies. When the device is falling, the accelerometer reports zero acceleration.

Now consider when you push the device along the surface of the table. Other forces are now in play such as the force being applied by your hand and the force due to friction between the device and the table. The `Accelerometer` measures all of these forces.

> **NOTE** In the Windows Phone 7.0 SDK, the Accelerometer class didn't inherit from SensorBase and was updated in the 7.1 SDK to conform to the SensorBase API. To avoid breaking changes between 7.0 and 7.1, Microsoft left the original API intact. The obsolete members include the accelerometer's `ReadingChanged` event and the `AccelerometerReadingEventArgs` class.

When a user shakes a phone, the x, y, and z acceleration values will rapidly change from one extreme to another in a fairly random pattern. By examining the x, y, and z values of the accelerometer, and how they change from one reading to the next, you can determine whether the device is in motion, and how the device is being held. Before we get into the details about exactly what the accelerometer measures and what the reported values mean, you'll hook up the sensor to the bars displayed in the user interface of your application.

8.3.1 Hooking up the sensor

The sample application you built in the previous section is designed to show how the data returned by the sensors changes as the user moves the phone. To see how the accelerometer data changes, you just need to read the `CurrentValue` property from an `Accelerometer` instance and update the three `Bar` controls allocated for the accelerometer's x, y, and z values. Before you can hook up a sensor, you need to declare a member field to reference the `Accelerometer` instance:

```
Accelerometer accelSensor;
```

In the `MainPage` constructor, initialize the field and set the `TimeBetweenUpdates`. You only set the property if the sensor is supported by the phone, so you check the `IsSupported` property. You set the `TimeBetweenUpdates` to match the tick interval of the `DispatchTimer` you use to trigger user interface updates:

```
if(Accelerometer.IsSupported)
{
    accelSensor = new Accelerometer();
    accelSensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
}
```

Next you start the sensor in the `start_Click` method. Add the following snippet right before the line where the timer's `Start` method is called:

```
if (Accelerometer.IsSupported)
{
    accelSensor.Start();
    runningMessage += "Accelerometer ";
}
```

You're adding the string *Accelerometer* to the message you display in the user interface, informing the user that the accelerometer was started. You only start the sensor if it's supported.

You also must stop the sensor in the `stop_Click` method when the Stop button is clicked:

```
if (Accelerometer.IsSupported)
    accelSensor.Stop();
```

The final step is to read the accelerometer data when the timer ticks and the `timer_Tick` event handler is called. You're going to isolate the code that reads the accelerometer data into a method named `ReadAccelerometerData`. The timer tick method calls the `ReadAccelerometerData` method, which is shown in the next listing.

Listing 8.4 Reading acceleration

```
void timer_Tick(object sender, EventArgs e)
{
    ReadAccelerometerData();
}

private void ReadAccelerometerData()
{
    if (Accelerometer.IsSupported)
    {
        AccelerometerReading reading = accelSensor.CurrentValue;
        Vector3 acceleration = reading.Acceleration;
        accelX.Value = acceleration.X;
        accelY.Value = acceleration.Y;
        accelZ.Value = acceleration.Z;
    }
}
```

Get acceleration vector ①

② **Update user interface**

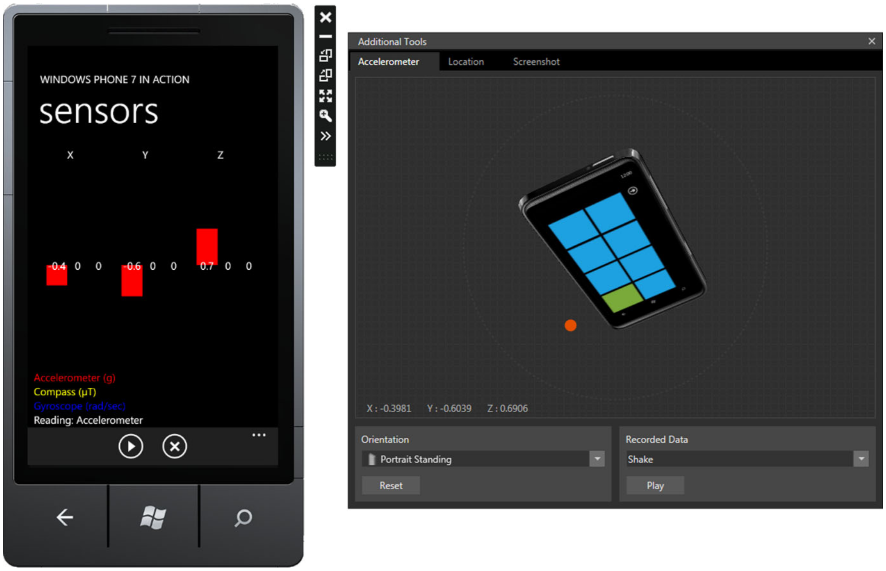
First you check whether the `Accelerometer` is supported before getting the current `AccelerometerReading` value from the sensor. Next you obtain the acceleration vector from the reading ①. The acceleration vector reports acceleration values in the three directions of the phone's coordinate system. Finally, you update the `Bar` controls in the user interface with the x, y, and z properties reported by the acceleration vector ②.

When you created the `Bar` controls for the accelerometer in MainPage.xaml, you set the `Scale` property to 100. The `Bar` controls are 400 pixels high allowing for positive and negative sections of 200 pixels each. The maximum value of the acceleration vector is ±2. Using this information, you can determine that the scale factor for the bar should be 100, or 200/2.

At this point, you should be able to run the application. If you run the application on a physical device, you should see the bars grow and shrink as you move the device about. Tilt it front to back, tilt it side to side, lay it down flat, hold it upside down. You can mimic all of these movements in the emulator using the accelerometer tool.

8.3.2 Acceleration in the emulator

With the Sensors sample application open in Visual Studio, run the application on the emulator and tap the application bar button labeled Start to begin collecting acceleration data. The emulator's default position is standing in portrait layout and the



Reading: Accelerometer

X : -0.3981 Y : -0.6039 Z : 0.6906

Orientation: Portrait Standing

Recorded Data: Shake

Figure 8.3 Controlling acceleration with the emulator's Accelerometer Tool

accelerometer reports an acceleration of -1 along the y axis. Open the Additional Tools windows using the expander button on the emulator's control bar. The Accelerometer Tool, shown in figure 8.3, is found in the first tab of the Additional Tools window.

The Accelerometer Tool allows you to move the device by dragging the orange dot. The device can also be changed from the Portrait Standing orientation to Portrait Flat, Landscape Standing, and Landscape Flat. The Accelerometer also plays a canned script that mimics shaking the device.

With the Sensors application running in the emulator, you should see the bars grow and shrink as you move the device about with the orange dot. Play the Shake script and watch how the acceleration bars bounce up and down as the data changes. Now that you have a better idea of the numbers reported by the `Accelerometer`, let's take a closer look at exactly what the numbers mean.

8.3.3 Interpreting the numbers

The accelerometer sensor in the phone senses the acceleration due to forces applied to the phone but ignores the acceleration due to gravity. When a device is at rest lying on a table, the table is exerting a force on the device that offsets the pull of gravity. The accelerometer measures the acceleration of the force that the table applies to the device. When the device is falling, the accelerometer reports zero acceleration. Figure 8.4 demonstrates the values reported by the accelerometer when a phone is held in various positions.

If the number reported by the device is related to the force a surface exerts on the device, why is the number negative instead of positive? Remember that the number

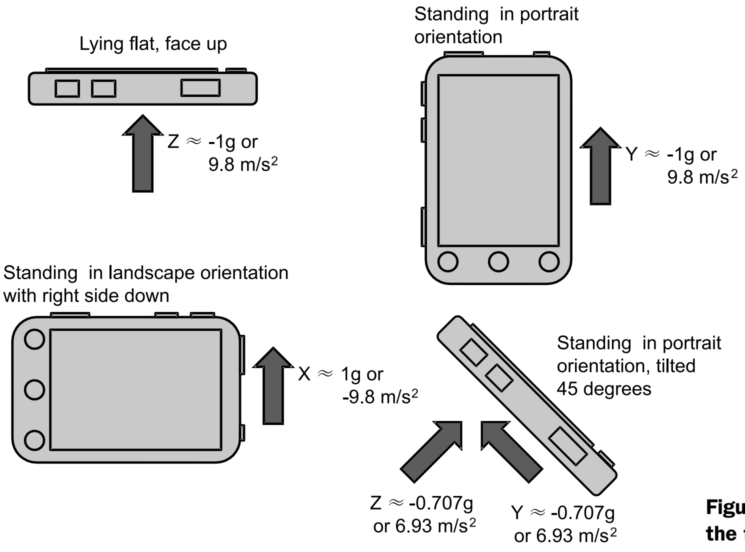

Lying flat, face up
Z ≈ -1g or 9.8 m/s2
Standing in portrait orientation
Y ≈ -1g or 9.8 m/s2
Standing in landscape orientation with right side down
X ≈ 1g or -9.8 m/s2
Standing in portrait orientation, tilted 45 degrees
Z ≈ -0.707g or 6.93 m/s2
Y ≈ -0.707g or 6.93 m/s2

Figure 8.4 Acceleration from the forces on a device at rest

reported is in terms of g or gravity, and g equals -9.8 m/s^2, a negative number. When the accelerometer reports a -1 (or a vector 1 pointing down), it really means a vector with the value 9.8 pointing up. Table 8.2 lists the approximate x, y, z values reported by the `Accelerometer` when the device is at rest in various positions.

Table 8.2 Accelerometer readings with the device at rest

Position	X	Y	Z
In free fall	0g	0g	0g
Flat on back, lying on a surface	0g	0g	-1g or 9.8 m/s^2
Flat on face	0g	0g	1g or -9.8 m/s^2
Standing portrait, bottom is down	0g	-1g or 9.8 m/s^2	0g
Standing portrait, top is down	0g	1g or -9.8 m/s^2	0g
Standing landscape, left is down	-1g or 9.8 m/s^2	0g	0g
Standing landscape, right is down	1g or -9.8 m/s^2	0g	0g

When you move a device, you apply a force along the direction you want the device to move. The force causes the device to accelerate, acceleration changes the velocity of the device, and it starts to move. Let's say your phone is resting flat face up on the surface of a table at point A. Now you give your device a modest push so that it slides along the surface to point B. The initial push is a moderate force in the positive X axis. After you release the device, allowing it to slide, your initial force stops, and the force due to friction begins to slow down the device until it stops moving. Figure 8.5 shows the values reported by the accelerometer in this scenario. The

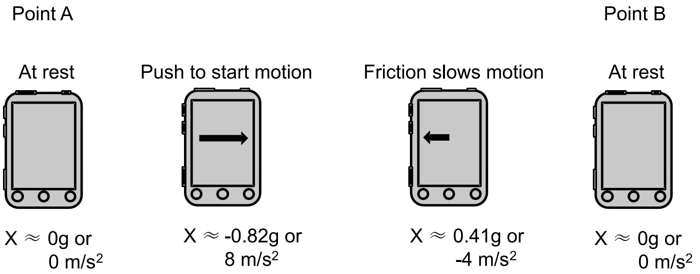


Figure 8.5 Acceleration due to motion

numbers are somewhat contrived, as real numbers will vary based on how hard the initial push is, and the amount of friction between the phone and the surface it's resting upon.

Again, note that the numbers reported by the accelerometer are opposite what you might expect. You push the device in the direction of the positive x axis, but the number reported is a negative value. Remember that the number reported is in terms of g or gravity, and g equals -9.8 m/s^2.

The figure demonstrates the forces involved with pushing a phone across a table. This is probably not something you do often. The same concepts can be applied when the device is moving in a user's hand. When motion begins, the user's hand is applying a force to the device in the direction of motion. When motion ends, the user's hand is applying force in the direction opposite the motion. When the user is moving the device, there may be a period between start and stop when the device is moving at a constant rate, and the acceleration in the direction of motion is zero.

By detecting changes in acceleration values, an application can determine when the device is being moved. The acceleration data can also tell you whether the device is being held flat, at an angle, or straight up and down. What the accelerometer can't tell you is which direction the device is pointed. If you need to know the direction the device is pointed, use the `Compass`.

8.4 Finding direction with the Compass

The `Compass` is useful when an application needs to know which direction a device is pointed relative to the real world. The direction is reported relative to the North Pole. This information is useful for applications such as the astronomy application we discussed earlier, where the application updates the display based on the direction the user is facing. The Compass is also useful in motion-sensitive applications that need to know when a device is rotated.

The `Compass` senses the strength and direction of the Earth's magnetic field. The `Compass` sensor decomposes the magnetic field into x, y, and z vectors, and reports the magnitude of the vectors in microteslas (μT). A tesla is the standard unit of measurement for a magnetic field.

The vertical intensity of the magnetic field (the z value when holding the device flat on a level surface) varies based on the latitude you're in. If you're near the equator, the vertical intensity will be nearly 0 µT. If you're in the far north (or south) the vertical intensity may be as high as 56 µT. The horizontal intensity varies inversely with latitude. At the equator, the horizontal intensity will be somewhere in the neighborhood of 32 µT. In the far north (or south) the horizontal intensity may be around 4 µT. The absolute intensity, which is the combination of horizontal and vertical intensity, varies as well. Figure 8.6 depicts the horizontal and vertical intensity vectors for a device held parallel to the Earth's surface.

The horizontal intensity vector points to magnetic north. The `Compass` is able to use this information to report heading, or the direction the device is pointing. The `Compass` reports heading compared to magnetic north and a true heading relative to the geographic north. The `Compass` reports heading as the number of degrees the device is turned clockwise from north.

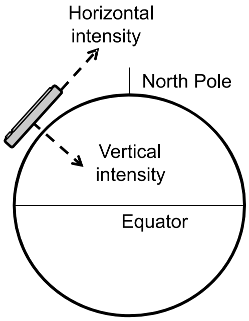


Figure 8.6 The vertical intensity vector points down into the Earth, whereas the horizontal intensity vector points at the magnetic North Pole.

> **NOTE** The Compass API was introduced in the Windows Phone 7.1 SDK. Many of the original 7.0 phones shipped with a compass but didn't ship with appropriate compass driver software. When the Windows Phone 7.5 update shipped, most phones also received updated drivers from the manufacturer to enable compass support. The Compass may not be supported on these phones since some phone models didn't receive new drivers, whereas other phones didn't successfully apply the driver update.

The `Compass` reports information with the `CompassReading` structure. The device direction is read with the `MagneticHeading` and `TrueHeading` properties. The magnetic intensity vectors are available from the `MagnetometerReading` property. Before we look closer at the `CompassReading`, you'll hook up the sensor to the bars displayed in the user interface of your application.

8.4.1 Hooking up the sensor

This section is going to look a whole lot like the section where you hooked up the accelerometer. You need to initialize the sensor in the constructor, start and stop it in the click event handlers, and create a method to read the sensor data. You start by defining a field in the `MainPage` class:

```
Compass compassSensor;
```

Next you initialize the sensor in the `MainPage` constructor. Before constructing the sensor, you first check whether the `Compass` is supported. After constructing the sensor you set the `TimeBetweenUpdates`:

```
if (Compass.IsSupported)
{
    compassSensor = new Compass();
    compassSensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
}
```

You start the sensor in the `start_Click` method:

```
if (Compass.IsSupported)
{
    compassSensor.Start();
    runningMessage += "Compass ";
}
```

You stop the sensor in the `stop_Click` method:

```
if (Compass.IsSupported)
    compassSensor.Stop();
```

Finally you create the `ReadCompassData` method in order to update the user interface with the sensor's `CurrentValue`. The following listing contains the implementation of the `ReadCompassData` method. Don't forget to call the new `ReadCompassData` method from the `timer_Tick` event handler.

Listing 8.5 Reading compass data

```
void ReadCompassData()
{
    if (Compass.IsSupported)
    {
        CompassReading reading = compassSensor.CurrentValue;
        Vector3 magnetic = reading.MagnetometerReading;

        compassX.Value = magnetic.X;
        compassY.Value = magnetic.Y;
        compassZ.Value = magnetic.Z;

        heading.Text = string.Format(
            "Compass (µT) : Heading {0} +/- {1} degrees",
            reading.TrueHeading, reading.HeadingAccuracy);
    }
}
```

➊ **Get reading**

➋ **Update user interface**

➌ **Report heading values**

You start by retrieving the current `CompassReading` value ➊ from the sensor and you assign the `MagnetometerReading` to a `Vector3` variable named `magnetic`. Next you update the user interface with the x, y, and z values of the `magnetic` vector ➋. Finally you update the message displayed near the bottom of the screen to show the values of the `TrueHeading` and `HeadingAccuracy` properties ➌. The `HeadingAccuracy` is the amount of potential error, in degrees, of the reported heading.

The `Scale` property of the compass-related bar controls declared in MainPage.xaml has been set to the value 4. The approximate maximum value from the `magnetic` vector is ±50 µT. Since the `Bar` controls are 400 pixels high and are divided into positive and negative halves, you set the `Scale` property to 4 or 200/50.

Now you're ready to run the application. You must run the application on a physical device because the `Compass` isn't supported on the emulator. When running the application, you should see the bars grow and shrink as you move the device about.

Can you figure out where north is by interpreting the numbers reported in the `Bar` controls?

8.4.2 *Interpreting the numbers*

We mentioned that the Earth's magnetic field can be represented as horizontal and vertical intensities. The `MagnetometerReading` represents the magnetic field as three different vectors, one aligned with each of the device's three axes. When you hold the device flat with the top of the device pointed to the north, the vertical intensity is aligned with the device's z axis, and the horizontal intensity is aligned with the y axis. If you spin the device so that it points to the northeast, the horizontal intensity is split between the x and y axes. Figure 8.7 shows the `MagnetometerReading` values as a device lying flat is pointed north, east, south, and west. The numbers in the figure are approximate for a location on the Earth's surface where the horizontal intensity is near 15 µT.

If you held the device standing vertical instead of flat, then the x and z axes would report the horizontal intensity, whereas the y axis would report the vertical intensity of the magnetic field. A device held at an angle relative to the ground would have a mixture of vertical and horizontal intensities reported along each axis.

No matter the angle the device is held, the heading properties will continue to report how far from north the device is pointed. The `HeadingAccuracy` should always have a small value. When the accuracy value grows, the sensor needs to be calibrated.

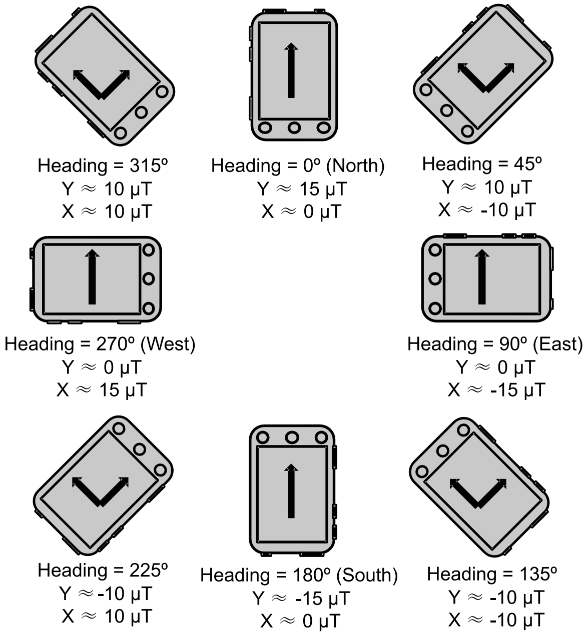

Heading = 315°
Y ≈ 10 µT
X ≈ 10 µT
Heading = 0° (North)
Y ≈ 15 µT
X ≈ 0 µT
Heading = 45°
Y ≈ 10 µT
X ≈ -10 µT
Heading = 270° (West)
Y ≈ 0 µT
X ≈ 15 µT
Heading = 90° (East)
Y ≈ 0 µT
X ≈ -15 µT
Heading = 225°
Y ≈ -10 µT
X ≈ 10 µT
Heading = 180° (South)
Y ≈ -15 µT
X ≈ 0 µT
Heading = 135°
Y ≈ -10 µT
X ≈ -10 µT

Figure 8.7 Examples of horizontal intensity vectors when lying flat

8.4.3 Calibrating the sensor

The Earth's magnetic field is relatively weak. Items in the local environment, such as strong magnets or large metal objects, will have an effect on the accuracy of the `Compass`. When the `HeadingAccuracy` is off by more than 20 degrees, the phone will raise the `Calibrate` event. When this happens, the phone wants the user to wave the device about in the air for a bit until it gets a grip on reality again.

You can ignore this event and nothing will happen, other than that the device will continue to report a large accuracy value. The event is only raised once for each instance of the `Compass`. If you'd like to be a responsible citizen, you should inform the user to wave the phone about in the air until the heading accuracy is more reasonable. You'll add support for the `Calibrate` event to your sample application.

In the `MainPage` constructor, subscribe to the `Calibrate` event right after the `Compass` is constructed:

```
compassSensor.Calibrate += compassSensor_Calibrate;
```

Now define the `Calibrate` event handler. Your sample application displays a message to the user asking them to wave the phone around in the air until the heading accuracy drops below 20 degrees. Other than displaying a message, the application can't do anything to help perform the calibration:

```
void compassSensor_Calibrate(object sender, CalibrationEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
        MessageBox.Show("The compass sensor needs to be calibrated.
➥ Wave the phone around in the air until the heading accuracy
➥ value is less than 20 degrees")
    );
}
```

The `CalibrationEventArgs` class doesn't define any properties and is just a simple derivation of the base `EventArgs` class.

The `Compass` is useful when an application needs to know which direction the device is pointed relative to the real world. If the device is turned or rotated, an application can determine how much the device was turned by comparing the current heading with a previous heading. The `Compass` isn't very useful if your application needs to be notified while the device is turning, or how quickly the device is turning. The `Gyroscope` is ideal for applications that respond when the device is rotated.

8.5 Pivoting with the Gyroscope

The `Gyroscope` sensor reports how quickly the device is turning on one or more of its axes. The rotational velocity is reported in radians per second, and when a device is rotated in a complete circle it'll rotate 2π radians or 6.28 radians. The values are reported with counterclockwise being the positive direction.

NOTE The gyroscope is optional hardware for Windows Phones and isn't supported on many phones. The `Gyroscope` class's `IsSupported` static property should be checked before using the gyroscope sensor.

The gyroscope only reports turning motion around an axis and if the device is held still, the sensor will report values of zero. If the device is moved from point A to point B without any twisting motion, the gyroscope will also report zero.

The `Gyroscope` reports values with the `GyroscopeReading` struct. Rotational velocities are read from the `GyroscopeReading` through the `RotationRate` property, a `Vector3` that breaks absolute movement into rotation about the x, y, and z axes. You'll now hook up the `Gyroscope` sensor to the user interface in your sample application so you can see the numbers for yourself.

8.5.1 Hooking up the sensor

The sensor APIs are intentionally similar, and hooking up the `Gyroscope` in your sample application is nearly identical to hooking up the `Accelerometer` and `Compass`. You start by declaring a field to reference the `Gyroscope` instance:

```
Gyroscope gyroSensor;
```

You then construct and initialize the field in the `MainPage` constructor:

```
if (Gyroscope.IsSupported)
{
    gyroSensor = new Gyroscope();
    gyroSensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
}
```

The sensor is started in the `start_Click` method:

```
if (Gyroscope.IsSupported)
{
    gyroSensor.Start();
    runningMessage += "Gyroscope ";
}
```

The sensor is stopped in the sample application's `stop_Click` method:

```
if (Gyroscope.IsSupported)
    gyroSensor.Stop();
```

As with the accelerometer and the compass sensors, you create a new method to read the `Gyroscope`'s `CurrentValue` and update the user interface. The new method is named `ReadGyroscopeData`, and is called from the `timer_Tick` method. The code for `ReadGyroscopeData` is shown in the following listing.

Listing 8.6 Reading gyroscope data

```
void ReadGyroscopeData()
{
    if (Gyroscope.IsSupported)
    {
            GyroscopeReading reading = gyroSensor.CurrentValue;
            Vector3 rotation = reading.RotationRate;

            gyroX.Value = rotation.X;
            gyroY.Value = rotation.Y;
            gyroZ.Value = rotation.Z;
        }
    }
```

❶ **Get reading**

❷ **Update user interface**

You start by retrieving the current `GyroscopeReading` value ❶ from the sensor and assign the `RotationRate` to a `Vector3` variable named `rotation`. Next you update the user interface with the x, y, and z values of the `rotation` vector ❷.

When you created the `Bar` controls for the gyroscope in MainPage.xaml, you set the `Scale` property to 32. The positive and negative bars are each 200 pixels each. You assume the maximum rotation rate is a full spin once per second or ±2π radians per second. With 2π equal to approximately 6.25, you calculated the scale of the `Bar` control at 32 or 200/6.25.

What can you do to see the gyroscope bars move in the application? Let's get dizzy. Do you have a spinning office chair? If so, you can hold the device flat in your hand and spin back and forth in your chair. You should see the Z-bar move up and down as you spin. Another example is to hold the device in your hand so that it's standing up in portrait mode. Now tilt the phone back until it's lying flat in your hand. You should see the X-bar move down and report a negative value. Tilt the phone back up, and the bar should move up and report a positive value.

We're now finished with the Sensors sample application. You've seen how each of the hardware sensors is exposed by classes and structures in the Sensors API. The sensors each return individual sets of data that can be used in various ways to build interesting applications. Each of the sensors tell you different bits of information about how the device is held, how the device is moving, and which direction the device in pointed in. Correlating this information across sensors can be tricky, and involves a solid understanding of physics, mathematics, and three-dimensional coordinate spaces. The Windows Phone SDK provides the `Motion` class to perform these calculations for you.

8.6 Wrapping up with the motion sensor

Unlike the other sensors we've covered so far in this chapter, the motion sensor isn't a hardware-based sensor. The motion sensor, represented in the Windows Phone SDK as the `Motion` class, is a wrapper around the `Accelerometer`, `Compass`, and `Gyroscope`. Instead of sensing data from hardware, the motion sensor consumes data from the other sensors and performs some convenient number crunching.

> **NOTE** The `Motion` class is supported if a phone has an accelerometer and compass. The motion sensor is supported even if a phone doesn't have gyroscope hardware installed. When the gyroscope isn't installed, the data provided by the Motion class may not be as accurate as the data provided when the gyroscope is present.

The `Motion` class analyzes the data provided by the `Accelerometer`, `Compass`, and `Gyroscope`. The `Motion` class reports the results of its data analysis in the `MotionReading` class. The `Motion` class separates motion-based acceleration from gravity. Motion-based acceleration is reported in `MotionReading`'s `DeviceAcceleration` property, whereas the `Gravity` property reports acceleration due to gravity. The `DeviceRotationRate` property reports rotational velocities obtained from the gyroscope.

The `Motion` class also provides tools for mapping device coordinates into real-world coordinates with the `AttitudeReading` class. An instance of the `AttitudeReading` class is returned from the `Attitude` property of the `MotionReading` class. The `AttitudeReading` class reports `Yaw`, `Pitch`, and `Roll` values. The `AttitudeReading` class also provides both a rotation `Matrix` and a `Quaternion` that can be used for coordinate mapping. We'll show you how to use the `AttitudeReading` to map coordinates in a new `MotionSensor` sample application.

Next you'll create a new sample application to demonstrate how the motion sensor works and how to use the numbers provided by the `MotionReading` class.

8.6.1 Building a motion enabled sample application

You're going to create a new sample application to show off the motion sensor. This new sample application is similar to the Sensors sample application you just finished. Create a new Windows Phone Application named MotionSensor, and add references to the Microsoft.Devices.Senors.dll and Microsoft.Xna.Framework.dll assemblies. The MotionSensor application will also use the `Bar` control you created in the Sensors application. Copy the Bar.xaml and Bar.xaml.cs files into the MotionSensor project and change the `Bar` class's namespace to `MotionSensor`.

MainPage.xaml is assembled similarly to the MainPage.xaml.cs used in the Sensors application. It's similar enough that you may wish to start by copying the markup for the Sensors application's `ContentPanel` and modifying it. The next listing shows the markup for the MotionSensor application's `ContentPanel`.

Listing 8.7 MainPage markup for the motion sensor sample application

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Grid.RowDefinitions>
    <RowDefinition Height="25" />
    <RowDefinition Height="400" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
```

① **Fourteen columns**

```
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
  </Grid.ColumnDefinitions>

  <TextBlock Text="X" Grid.Column="1" />
  <TextBlock Text="Y" Grid.Column="6" />
  <TextBlock Text="Z" Grid.Column="11" />

  <l:Bar x:Name="accelX" Grid.Row="1" Grid.Column="0"
    BarFill="Red" Scale="67" />
  <l:Bar x:Name="accelY" Grid.Row="1" Grid.Column="5"
    BarFill="Red" Scale="67" />
  <l:Bar x:Name="accelZ" Grid.Row="1" Grid.Column="10"
    BarFill="Red" Scale="67" />

  <l:Bar x:Name="gravityX" Grid.Row="1" Grid.Column="1"
    BarFill="Yellow" Scale="200" />
  <l:Bar x:Name="gravityY" Grid.Row="1" Grid.Column="6"
    BarFill="Yellow" Scale="200" />
  <l:Bar x:Name="gravityZ" Grid.Row="1" Grid.Column="11"
    BarFill="Yellow" Scale="200" />

  <l:Bar x:Name="gyroX" Grid.Row="1" Grid.Column="2"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroY" Grid.Row="1" Grid.Column="7"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroZ" Grid.Row="1" Grid.Column="12"
    BarFill="Blue" Scale="32" />

  <l:Bar x:Name="attitudeX" Grid.Row="1" Grid.Column="3"
    BarFill="Violet" Scale="64" />
  <l:Bar x:Name="attitudeY" Grid.Row="1" Grid.Column="8"
    BarFill="Violet" Scale="128" />
  <l:Bar x:Name="attitudeZ" Grid.Row="1" Grid.Column="13"
    BarFill="Violet" Scale="32" />

  <StackPanel Grid.Row="2" Grid.ColumnSpan="14">
    <TextBlock Foreground="Red" Text="Acceleration (g)" />
    <TextBlock Foreground="Yellow" Text="Gravity (g)" />
    <TextBlock Foreground="Blue" Text="Rotation (rad/sec)" />
    <TextBlock x:Name="point" Foreground="Violet" Text="Attitude" />
  </StackPanel >
</Grid>
```

① **Fourteen columns**

② **Three bars for each data point**

③ **Legend and messages**

You start by dividing the `ContentPanel` into three rows and fourteen columns ①. The first row contains three `TextBlocks` serving as the titles for the x, y, and z coordinates. The second row shows three bars ② for each of the linear acceleration, gravity, rotation, and attitude readings. Allowing for three columns for each reading, and two spacer columns, you need a total of fourteen columns, as displayed in figure 8.8. The last row ③ contains a legend and messages.

When setting the scale factors for the acceleration, gravity, and rotation bars, you assume the maximum values returned by the sensor are 3, 1, and 2π respectively. The scale values for the attitude bars reflect that the maximum values for `Pitch`, `Roll`, and `Yaw` are π, π/2 and 2π respectively. We'll examine each of the readings values in more detail in section 8.6.3.

The MotionSensor sample application also declares Start and Stop buttons on the application bar. Copy the application bar markup from the Sensors application. Don't forget to create the Images folder and copy the SDK icons.

The application isn't quite ready to run and doesn't compile yet. The application is missing the `Click` event handlers for the two application bar buttons. You'll implement the two missing methods after you hook up the motion sensor.

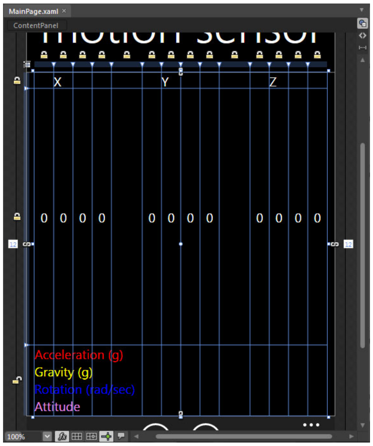


Figure 8.8 A screen shot of the ContentPanel in MainPage.xaml as displayed by the designer view in Microsoft Expression Blend

8.6.2 Hooking up the sensor

In the Sensors application you polled all three sensors using a `DispatchTimer` and the `CurrentValue` property of each sensor. In this sample application you're going to use the `CurrentValueChanged` event instead. Start by creating a class field for the motion sensor. Don't forget to add using statements for `Microsoft.Devices.Sensors`:

```
Motion sensor;
```

The sensor variable is constructed and initialized in the `MainPage` constructor. In addition to constructing the sensor, you set the `TimeBetweenUpdates` property and subscribe to the `CurrentValueChanged` and `Calibrate` methods. The next listing shows the implementation of the `MainPage` constructor.

Listing 8.8 Initializing the motion sensor

```
public MainPage()
{
    InitializeComponent();

    if (Motion.IsSupported)
    {
        sensor = new Motion();
        sensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
        sensor.CurrentValueChanged += sensor_CurrentValueChanged;
        sensor.Calibrate += sensor_Calibrate;
    }
}
```

① Construct motion sensor

② Subscribe to value changed event

Before attempting to use the `Motion` class, you check the `IsSupported` static property to see whether the motion sensor is supported by the device running the application. Once you're sure that the motion sensor is supported you construct a new instance of the `Motion` class ①. You then set the `TimeBetweenUpdates` property to 66 milliseconds so that you receive updates approximately 17 times each second. Next you subscribe to the `CurrentValueChanged` event ②, registering the method `sensor_CurrentValueChanged` as the event handler. Finally, you subscribe to the `Calibrate` event.

Because the `Motion` class wraps the compass sensor, you need to be aware of when the compass needs calibration. Copy the `Calibrate` event handler implementation from the Sensors program into a method named `sensor_Calibrate`.

Let's examine the `CurrentValueChanged` event handler. The `Motion` class raises the `CurrentValueChanged` event on a background thread. If the user interface is updated from the event handler, the update code must be executed on the UI thread. The next listing shows the implementation of the `sensor_CurrentValueChanged` method.

Listing 8.9 Handling the `CurrentValueChanged` event

```
void sensor_CurrentValueChanged(object sender,
    SensorReadingEventArgs<MotionReading> e)
{
    MotionReading reading = e.SensorReading;
    Dispatcher.BeginInvoke(() =>
    {
        Vector3 acceleration = reading.DeviceAcceleration;
        accelX.Value = acceleration.X;
        accelY.Value = acceleration.Y;
        accelZ.Value = acceleration.Z;

        Vector3 gravity = reading.Gravity;
        gravityX.Value = gravity.X;
        gravityY.Value = gravity.Y;
        gravityZ.Value = gravity.Z;

        Vector3 rotation = reading.DeviceRotationRate;
        gyroX.Value = rotation.X;
        gyroY.Value = rotation.Y;
        gyroZ.Value = rotation.Z;

        AttitudeReading attitude = reading.Attitude;
        attitudeX.Value = attitude.Pitch;
        attitudeY.Value = attitude.Roll;
        attitudeZ.Value = attitude.Yaw;

        Vector3 worldSpacePoint = new Vector3(0.0f, 10.0f, 0.0f);
        Vector3 bodySpacePoint =
            Vector3.Transform(worldSpacePoint, attitude.RotationMatrix);
        point.Text = string.Format("Attitude: Transform of (0.0, 10.0, 0.0)
➥ = ({0:F1}, {1:F1}, {2:F1})",
            bodySpacePoint.X, bodySpacePoint.Y, bodySpacePoint.Z);
    });
}
```

① **Work on UI thread**

② **Update bar controls**

③ **Transform coordinate point**

The `CurrentValueChanged` event is raised on a background thread. In your event handler implementation, you update the user interface and use the `Dispatcher` object to run your code on the user interface thread ①. The next section of the listing reads various properties of the provided `MotionReading` instance and updates the twelve

`Bar` controls ❷. Finally, you use the `AttitudeReading`'s `RotationMatrix` to convert a real-world coordinate to the coordinate system of the phone ❸.

You still need to create the `Click` event handlers for the Start and Stop application bar buttons. The `start_Click` method is simple. The method checks whether the `Motion` class is supported, and then calls the `Start` method:

```
private void start_Click(object sender, EventArgs e)
{
    if (Motion.IsSupported)
        sensor.Start();
}
```

The `stop_Click` method is just as simple:

```
private void stop_Click(object sender, EventArgs e)
{
    if (Motion.IsSupported)
        sensor.Stop();
}
```

With the two click event handlers implemented, the code for the sample application is complete. Deploy the application to your motion-sensor-enabled device and run the application. Start the sensor and examine the acceleration, gravity, rotation, and attitude values as you move your phone around. Don't forget to look at the attitude message line and how the coordinate point is transformed. Let's take a closer look at the readings reported by the `Motion` class and how to interpret the numbers.

8.6.3 *Interpreting the numbers*

We talked about acceleration earlier in the chapter, and how the data reported by the `Accelerometer` mixes both the acceleration due to gravity and the acceleration due to motion. There are many scenarios where it's important to separate the two types of acceleration. Consider an application or game that works by tilting the phone, such as one that moves a ball through a maze. This type of application is only interested in the acceleration due to gravity. Without the convenient `Gravity` property of the `MotionReading`, the maze application would have to somehow account for the acceleration due to motion. We demonstrate how to use the `Gravity` vector to control a simple game in chapter 15.

The other convenience provided by the `Motion` class is the calculation of the `Attitude` data. To understand the `Attitude` you need to understand the frame of reference, or coordinate system for both the real world and the device. The `Motion` class assumes a real-world coordinate system where y points due north, z points straight up, and x points due east. When the device is lying flat, face up, with the top of the device pointing north, the device's frame of reference matches the real world frame of reference. This is shown in figure 8.9.

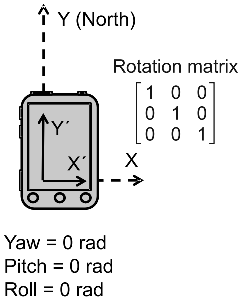


Figure 8.9 The device frame aligned with the world frame when the device is lying flat and pointed North

The `Yaw`, `Pitch`, and `Roll` readings are all approximately zero, and the rotation matrix is the identity matrix. An object at point (0, 10, 0) in the world frame will have the same coordinates in the device frame. The device's y axis, labeled Y' in the figure, is pointing north and the x axis, labeled X', is pointing east.

When the device is rotated, its frame of reference no longer matches the real world frame of reference. If the top of the device lying flat is rotated to point east, the device is considered to be rotated 270 degrees. The attitude reading will have a `Yaw` reading of 3/2 π radians, or 270 degrees. The `Yaw`, rotation about the z axis, is read as the counterclockwise angle between the two y axes. This is shown in figure 8.10.

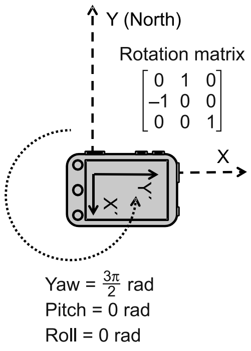


Figure 8.10 The device rotated 3/2 π radians or 270 degrees around the z axis

Now the device's y axis is pointing east and the x axis is pointing south. Again, consider an object at the coordinate (0, 10, 0) in the world frame. This same object will have the coordinates (-10, 0, 0) in the device frame.

With the top of the device still pointed east, raise the top of the device until it's in the standing portrait orientation with the back of the device facing east. This is shown in figure 8.11. In this case you've rotated the device frame about the x axis and changed the `Pitch` of the device. The attitude reading will still have a `Yaw` reading of 3/2 π radians, but will now also have a `Pitch` reading of 1/2 π radians, or 90 degrees. The `Pitch`, or rotation about the x axis, is read as a counterclockwise angle.

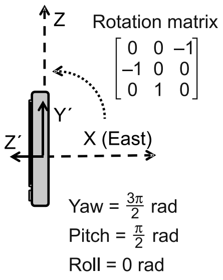


Figure 8.11 The device rotated 270 degrees around the z axis and 90 degrees around the x axis

Now the device's y axis is pointing up toward the sky, aligned with the world frame's z axis. The device's z axis is pointing to the west. The device's x axis is still pointing south. Again, consider an object at the coordinate (0, 10, 0) in the world frame. This same object will still have the coordinates (-10, 0, 0) in the device frame because changing the pitch didn't change the direction of the device's x axis.

When working with the `AttitudeReading`, you must remember that the `Yaw`, `Pitch`, and `Roll` values are order-dependent. To translate a point in one frame of reference to a point in another frame of reference, you must apply `Yaw` first, followed by `Pitch`, then by `Roll`.

Though we've referred to the `Motion` class as a motion sensor, it's really more of a service than a sensor. The `Motion` class makes use of a few different sources of data to provide a convenient service for detecting motion and position.

8.7 Summary

In this chapter we've covered three different hardware sensors, and one class that wraps the other sensors. The `Accelerometer` reports acceleration due to the forces acting on a device. The `Compass` reports the strength of local magnetic fields as well as the heading of the device relative to north. The `Gyroscope` reports the rotational velocity of the device. There aren't any sensors that report linear velocity or rotational acceleration. There is also not a sensor that reports exactly how far a phone has moved.

The `Motion` sensor uses data from the `Accelerometer`, `Compass`, and `Gyroscope` to perform a few complex calculations to separate acceleration into gravity and motion components. The motion sensor also provides information necessary to convert device coordinates into real-world coordinates.

Application developers should consider mixing one or more sensors with the location service to build applications that mesh real world with the digital world. Novel augmented reality applications can be built to show the user the location of nearby landmarks or the position of constellations in the night sky.

In the next chapter, we'll explore the networking features of the Windows Phone SDK. You'll learn how to determine network connection state and how to connect to web services. You'll also learn how to send notifications to a phone from a web service.

9

Network communication with push notifications and sockets

This chapter covers

- Detecting network information
- Pushing notifications
- Updating application tiles
- Communicating with sockets

In the past few years, the way that most mobile users use their phones has dramatically changed. The mobile phone has become our constant companion and helps keep us connected with family, friends, and coworkers. We use the phone to read news, check emails/twitter/Facebook, post updates, and play games. Most of the applications that you're using on your phone have one thing in common. Can you guess what it is? All those applications access the network to retrieve the information stored in the cloud. Consuming resources from the network is crucial even for a normal mobile user. As a developer who is developing an application for mobile devices, you need to understand how to efficiently consume network resources.

Networked applications need to know whether a network connection is available, and if so, what type of connection is enabled. We open up this chapter by discussing how you can detect the network connection before consuming resources. We show you how to detect whether the device is connected to a cellular data network or a Wi-Fi access point, or whether the device is in airplane mode.

To keep a user informed, applications need to retrieve updates from the internet, but how can they do that when they're not running? You learned about background agents in chapter 3, which require some code to be running on the phone, periodically checking for updates. In this chapter we explore another technology—push notifications. With push notifications, code runs in a web service and updates are pushed to the phone. You'll learn how a web service can push notifications to a phone. Notifications are used to update an application's tile or display a toast message to the user.

While we explore push notifications, you'll learn about the `HttpWebRequest` class that's useful for connecting to and sharing data over an HTTP connection. HTTP connections are just one type of network connection. The Windows Phone 7.1 SDK also provides a `Socket` class that enables lower-level network communication with TCP and UDP.

To demonstrate push notifications and networking, you're going to build three sample applications:

- A push notification simulator that will demonstrate how to detect network status and how to push notifications to a Windows Phone
- A push notification client that will register for and receive push notifications from the simulator
- A simple chat application using TCP sockets to communicate with a chat server

All three applications require a network connection. We open the chapter with a look at the network information classes and how to use them to detect network connectivity status.

9.1 Detecting network connectivity

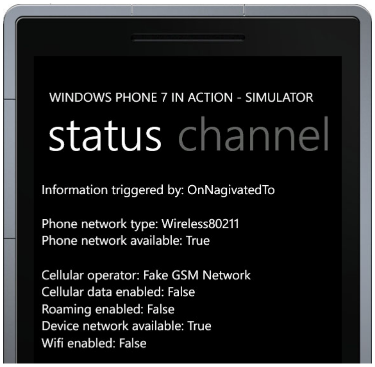
Detecting network connectivity is a crucial task for mobile application developers because many of the interesting mobile applications require access to network resources. Applications that utilize the network capabilities should be aware of the status of the network connection and whether that network is using the cellular data network or a Wi-Fi network. Mobile applications must also adjust to changes in the network connection, which can alternate from no access to cellular data and Wi-Fi in a short period of time. The Windows Phone SDK contains a set of network information APIs unique to the Windows Phone in the `Microsoft.Phone.Net.NetworkInformation` namespace.

Before we look at `Microsoft.Phone.Net.NetworkInformation`, you'll create the Visual Studio project for your first sample application, the push notification simulator. You'll use the simulator application to demonstrate networking code. Using the Windows Phone Application template, create a new project named NotificationSimulator. When creating the project, name the solution PushNotifications. The simulator sample application is built using a `Pivot` control. Delete MainPage.xaml and add a new `MainPage` using the Windows Phone Pivot Page item template. The first pivot in the application, shown in figure 9.1, contains a single `TextBlock` that's used to display network connection status.

The `TextBlock` will display a formatted string built up from network and device information you gather from a couple of different APIs. Give the `TextBlock` the name `statusMessage`:

```
<controls:PivotItem Header="status">
    <TextBlock x:Name="statusMessage" />
</controls:PivotItem>
```

This sample application will display the network availability in the `statusMessage` `TextBlock` whenever the application is activated. Create a method named `LoadInformation` in MainPage.xaml.cs that will be called to gather up network information and display it in the user interface:



WINDOWS PHONE 7 IN ACTION - SIMULATOR

status channel

Information triggered by: OnNagivatedTo

Phone network type: Wireless80211
Phone network available: True

Cellular operator: Fake GSM Network
Cellular data enabled: False
Roaming enabled: False
Device network available: True
Wifi enabled: False

Figure 9.1 The Networking sample application

```
private void LoadInformation(string trigger)
{
    string information = string.Format(
        "Information triggered by: {0}", trigger);
    statusMessage.Text = information;
}
```

Now call the `LoadInformation` method from the `OnNavigatedTo` method so that the network status is displayed whenever the page is activated:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    LoadInformation("OnNagivatedTo");
}
```

At this point, the sample application doesn't do anything interesting, other than display *OnNavigatedTo* in the user interface. You'll now liven up the application by displaying device settings to the user.

9.1.1 Reading device settings

The `Microsoft.Phone.Net.NetworkInformation` namespace was created specifically for the phone and has everything you need for reading network information. The namespace contains two interesting classes named `NetworkInterface` and `DeviceNetworkInformation`.

Table 9.1 Properties of DeviceNetworkInformation

Name	Description
`CellularMobileOperator`	Returns the name of the wireless service provider.
`IsCellularDataEnabled`	Returns `true` if the cellular data network is available. Returns `false` if airplane mode is enabled.
`IsCellularDataRoamingEnabled`	Returns `true` if the data roaming option is set to roam, and `false` if the option is set to don't roam.

Table 9.1 Properties of DeviceNetworkInformation *(continued)*

Name	Description
`IsNetworkAvailable`	Returns `true` if any network connection is available.
`IsWiFiEnabled`	Returns `true` if Wi-Fi network is enabled in the settings application, even if the device isn't connected to a Wi-Fi network.

The `DeviceNetworkInformation` class is new to the Windows Phone 7.1 SDK and is only available to applications written for the Windows Phone 7.5 operating system. `DeviceNetworkInformation` provides several properties, listed in table 9.1, that report the current values of networking-related options specified in the settings application.

Update the NotificationSimulator sample application to display the properties of the `DeviceNetworkInformation` class. You start by adding a call to a new `LoadDeviceInformation` method in the `LoadInformation` method you created in the last section:

```
private void LoadInformation(string trigger)
{
    string information = string.Format(
        "Information triggered by: {0}\n\n{1}",
        trigger, LoadDeviceInformation());
    statusMessage.Text = information;
}
```

The new `LoadDeviceInformation` method, shown in the following listing, returns a formatted string containing a description and value for each of the properties of the `DeviceNetworkInformation` class.

Listing 9.1 Reading `DeviceNetworkInformation` properties

```
private string LoadDeviceInformation()
{
    return string.Format("Cellular operator: {0}\n" +
        "Cellular data enabled: {1}\nRoaming enabled: {2}\n" +
        "Device network available: {3}\nWi-Fi enabled: {4}\n",
        DeviceNetworkInformation.CellularMobileOperator,
        DeviceNetworkInformation.IsCellularDataEnabled,
        DeviceNetworkInformation.IsCellularDataRoamingEnabled,
        DeviceNetworkInformation.IsNetworkAvailable,
        DeviceNetworkInformation.IsWiFiEnabled);
}
```

The `DeviceNetworkInformation` class provides an event named `NetworkAvailabilityChanged` that reports network status changes such as connecting, disconnecting, and changing the roaming status. You'll register for this event in the `MainPage` constructor and use it as a trigger for updating the user interface:

```
DeviceNetworkInformation.NetworkAvailabilityChanged +=
    (sender, e) => LoadInformation(e.NotificationType.ToString());
```

You pass the value of the `NotificationType` property as the trigger parameter to `LoadInformation`.`NotificationType` is an enum with the value `InterfaceConnected`, `InterfaceDisconnected`, or `CharacteristicUpdate`.

`DeviceNetworkInformation` tells you about the phone's network-related settings. Next we look at the `NetworkInterface`, which tells you not only whether a network is connected, but the type of network you're connected to.

9.1.2 Using the NetworkInterface class

If you're coming from a Silverlight or .NET background, you might already be familiar with the `NetworkInterface` class in the `System.Net.NetworkInformation` namespace, which is used to determine network connectivity. The `Microsoft.Phone.Net.NetworkInformation` namespace provides a phone-specific implementation of the `NetworkInterface` class, also named `NetworkInterface`. The `NetworkInterface` class is inherited from the `System.Net.NetworkInformation.NetworkInterface` class and both classes have a static method named `GetIsNetworkAvailable`. The phone's `NetworkInterface` class also provides an enum property named `NetworkInterfaceType`, which can be used to determine if the current connection is on a cellular data or Wi-Fi network. The list of possible values for `NetworkInterfaceType` is listed in table 9.2.

Table 9.2 List of network types

Member Name	Description
`None`	The device is not connected to a network.
`Ethernet`	The device is connected to a wired Ethernet network. Usually this indicates the device is connected to a desktop computer and is sharing the desktop's network connection.
`Wireless80211`	The device is connected to a Wi-Fi network.
`MobileBroadbandGsm`	The device is connected to a GSM cellular network.
`MobileBroadbandCdma`	The device is connected to a CDMA cellular network.

The `NetworkInterfaceType` enumeration defines approximately two dozen other network types, which aren't used by the Windows Phone 7.5 operating system. Only the five network types that are mentioned in table 9.2 are currently supported by the phone. When your mobile device is connected to the computer, the reported network type will be `Ethernet`. If you're on a Wi-Fi network, the network type will be `Wireless80211`, and if you're using a cellular data network, the network type will be `MobileBroadbandGsm` or `MobileBroadbandCdma`.

Next you'll wire in the data returned from `NetworkInterface` into the user interface. You start by adding a new method named `LoadPhoneNetworkInformation`:

```
private string LoadPhoneNetworkInformation()
{
    return string.Format(
        "Phone network type: {0}\nPhone network available: {1}\n",
        NetworkInterface.NetworkInterfaceType,
        NetworkInterface.GetIsNetworkAvailable());
}
```

It's important to know that calls to the `NetworkInterfaceType` property may take a long time to complete. When calling the property from your code, you should take extra care to not block the user interface thread. In your sample application, you'll update the `LoadInformation` method so that it runs on a background thread. The following listing shows the new implementation of `LoadInformation`.

Listing 9.2 Loading network information on a background thread

```
private void LoadInformation(string trigger)
{
    statusMessage.Text = "loading...";
    ThreadPool.QueueUserWorkItem((state) =>
    {
        string information = string.Format(
            "Information triggered by: {0}\n\n{1}\n{2}",
            trigger, LoadPhoneNetworkInformation(),
            LoadDeviceInformation());

        Dispatcher.BeginInvoke(() => statusMessage.Text = information);
    });
}
```

① Spin up background thread

② Update user interface

The first change you implement is to display a loading message in the user interface. Next you wrap the existing code in an anonymous function and use the `QueueUserWorkItem` method to request that the `ThreadPool` class execute the function on a background thread ①. Once the information string is built, you use the `Dispatcher` to update the `statusMessage` TextBlock ② on the user interface thread.

Run the NotificationSimulator application and examine the data reported on the screen. Change your device's setting by turning on airplane mode or disabling Wi-Fi. Does the screen update as you expect? In the next section you'll add more features to the NotificationSimulator application as you learn how to push notifications to the phone.

9.2 Pushing notifications to a phone

Distributed application developers have two options when notifying phone users with updates and alerts. The first requires code on the phone to periodically pull data from a web service, and can be implemented using background agents. We covered background agents in chapter 3. An alternative model is for a web service to push notifications to a phone. Microsoft provides the Push Notification Service (MPN) to enable sending updates and alerts to a phone, without an active agent running on the phone. The Microsoft Push Notification Service and Windows Phone support three different types of notifications.

In this section we'll explore the different types of notifications and how to use the MPN service to send them. Before remote services can send a message to a phone, the user must register the phone with the service and we'll show you the API calls used for registration. Finally, we'll update the simulator application to send all three types of push notifications.

9.2.1 Three types of notifications

Windows phone provides three types of push notifications that you can use to send a notification from a web service to a Windows Phone. These three types are named *toast*, *tile*, and *raw* notifications:

- Toast notifications are short messages displayed at the top of the screen.
- Tile notifications update an application's tile that's pinned to the start screen.
- Raw notifications are used to send custom content to a running application.

Notifications are only sent to phones that have subscribed to a *notification channel*. Notification channels are linked to specific applications. Raw notifications are only displayed if the client application implements custom code to display the content of the message. Let's take a closer look at what the user sees when toast and tile notifications are received by a phone.

VIEWING A TOAST NOTIFICATION

A toast notification is made up of a title and short content message, as shown in figure 9.2. The user can dismiss the notification by flicking to the right. The user can tap the toast to launch the application. The application developer can define a custom launch URI as part of the toast.

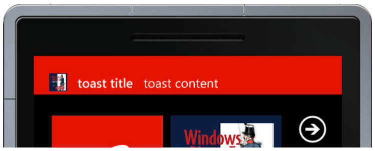


Figure 9.2 Toast notification

The toast will only be shown at the top of the screen if the linked application isn't running. When the application is running, the application is notified of the toast through a custom event. You'll learn how to use both the launch URI and the notification event later in the sample client application you build in this chapter.

BRINGING TILES TO LIFE WITH TILE NOTIFICATIONS

A tile notification is used to update the primary Application Tile on the start screen of the phone. If an application isn't pinned to the start screen, tile notifications won't have any effect. Tiles have both a front and a back. The Windows Phone start screen will flip the tile periodically, showing both front and back to the user. The front of the tile contains a title, count, and background image. The back of the tile has a title, a short content message, and a background image. The application tile from the sample client application is shown in figure 9.3.

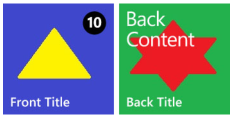


Figure 9.3 Title notification

The maximum value of the count property is 99. The background images should be 173 x 173 and other sized images will be stretched to fit. It's recommended that all tile images are shipped with the application, but images can be served by remote web servers.

9.2.2 Push notification workflow

Web services can't send notifications to any random Windows Phone. Users must first install an application on the phone, and the application code is then required to open a notification channel. The Windows Phone supports a limited number of open channels and there's no guarantee that an application will be able to open a new channel. Once the channel is opened, the application must forward the channel URI to its web service. Figure 9.4 shows the workflow for opening a notification channel and sending push notifications.

1. A push-enabled Windows Phone application is installed. When the application is run by the user, it opens a notification channel with the Microsoft Push Notification service. The MPN service will return a unique URI that a web service can use to push notifications to the phone.
2. Once the channel URI is returned to the application, the application sends the channel URI to the web service. The web service will use the channel URI when posting notifications to the MPN service.
3. At some point in the future something interesting happens, and the web service constructs notification messages to be sent to each phone that has sent its channel URI to the web server. The web service uses HTTP POST to ask the MPN service to send the notification to the phone's channel URI.
4. The MPN service forwards the notification message to the phone identified by the channel URI.

Since this is a book about programming Windows Phone, and not about web server programming, you're not going to build a web service. Instead you'll add features to the NotificationSimulator sample application so that it simulates a web service by performing HTTP POST calls directly to the MPN service. The NotificationSimulator application will be paired with a new NotificationClient application which will open the notification channel.

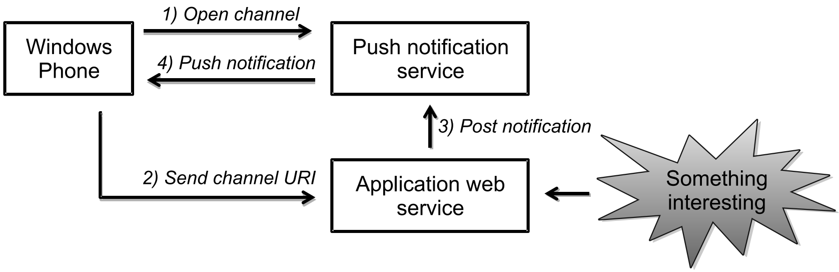


Figure 9.4 Push notification

9.2.3 Creating a Push Notification client

Create a new project in the PushNotifications solution using the Windows Phone Application project template. Just like you did earlier, delete the MainPage and add a new MainPage using the Windows Phone Pivot Page item template. You can see a screenshot of the completed client project in figure 9.5.

The XAML markup for the main page contains a `Pivot` control and an application bar. The next listing contains the markup for the first `PivotItem` in the `Pivot` control. You'll add a second pivot later in the chapter.

Listing 9.3 The markup for MainPage.xaml's `Pivot` control

```
<controls:PivotItem Header="notification">
  <StackPanel>
    <TextBlock Text="Channel Connection Status:"
      Style="{StaticResource PhoneTextTitle2Style}" />
    <TextBlock x:Name="channelStatus" Text="channel does not exist"
      TextWrapping="Wrap" Style="{StaticResource PhoneTextSmallStyle}"/>
    <TextBlock Text="Channel URI:"
      Style="{StaticResource PhoneTextTitle2Style}" />
    <TextBlock x:Name="channelUri" TextWrapping="Wrap"
      Style="{StaticResource PhoneTextSmallStyle}" />
    <TextBlock Text="Notification message:"
      Style="{StaticResource PhoneTextTitle2Style}" />
    <TextBlock x:Name="notificationMessage" Text="(no message)"
      TextWrapping="Wrap" Style="{StaticResource PhoneTextSmallStyle}"/>
  </StackPanel>
</controls:PivotItem>
```

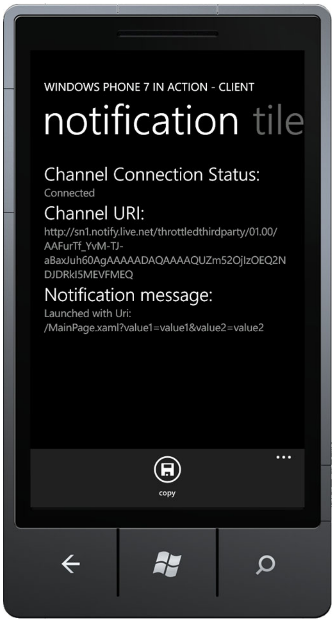
① **Display notification details**

The `PivotItem` displays a simple user interface with several `TextBlock`s. The `TextBlock`s are used either for labels or to display the channel's connection status and the channel URI. The last `TextBlock` ① will be used to display information from toast notifications that are received while the application is running.

The first time the user runs the application, push notifications aren't enabled. To enable the channel the first time the application is executed, and to reconnect on subsequent launches, you call a new method named `SetupChannel` inside the `MainPage` constructor:

```
public MainPage()
{
    InitializeComponent();
    SetupChannel();
}
```

Before you can implement `SetupChannel` you need to learn how to use the push notification APIs to open a notification channel.



Channel Connection Status:
Connected
Channel URI:
http://sn1.notify.live.net/throttledthirdparty/01.00/AAFurTf_YvM-TJ-aBaxJuh60AgAAAAADAQAAAAQUZm52OjIzOEQ2NDJDRkI5MEVFMEQ
Notification message:
Launched with Uri:
/MainPage.xaml?value1=value1&value2=value2

Figure 9.5 A screenshot of the completed notification client

9.2.4 Opening a notification channel

Push notification channels are created and opened through instances of the `HttpNotificationChannel` class, which is found in the `Microsoft.Phone.Notification` namespace. The `Microsoft.Phone.Notification` namespace contains other classes and enums used to support the methods and events defined by `HttpNotificationChannel`. We'll examine the supporting classes as you build the NotificationChannel sample application.

When an `HttpNotificationChannel` is constructed, it's given a name. The channel name is used by the client when looking for already opened channels. In MainPage.xaml.cs of the NotificationClient application, add a constant string for the channel name, as well as a field to reference an opened notification channel:

```
const string CHANNEL_NAME = "PushNotificationChannel";
HttpNotificationChannel channel;
```

Once a channel is constructed, the channel is opened using the `Open` method. Opening a channel isn't as easy as just constructing the channel and calling the `Open` method. Applications must check whether an open channel already exists. `HttpNotificationChannel` provides a static `Find` method to allow application code to look for existing open channels. You use the `Find` method in the `SetupChannel` method implementation, shown in the next listing.

Listing 9.4 Finding and constructing channels

```
void SetupChannel()
{
    bool newChannel = false;
    channel = HttpNotificationChannel.Find(CHANNEL_NAME);
    if (channel == null)
    {
        channel = new HttpNotificationChannel(CHANNEL_NAME);
        newChannel = true;
    }

    channel.ConnectionStatusChanged += channel_ConnectionStatusChanged;
    channel.ChannelUriUpdated += channel_ChannelUriUpdated;
    channel.ErrorOccurred += channel_ErrorOccurred;

    if (newChannel)
    {
        channel.Open();
        channel.BindToShellTile();
        channel.BindToShellToast();
    }

    channelStatus.Text = channel.ConnectionStatus.ToString();
    if(channel.ChannelUri != null)
        channelUri.Text = channel.ChannelUri.ToString();
}
```

① **Create new channel if not found**

② **Open and configure new channels**

③ **Update user interface**

You start by looking to see whether an open channel already exists, and only create a new `HttpNotificationChannel` ① when an existing channel isn't found. After you have a channel instance, you subscribe to the `ConnectionStatusChanged`, `ChannelUriUpdated`, and `ErrorOccurred` events. Next, you open the channel with the `Open` method, and configure ❷ the channel to receive tile and toast notifications by calling the two bind methods. If the bind methods aren't used, those types of notifications won't be sent to the phone. Though you bind them once while initializing your sample, a real application should provide user settings allowing the user to specify which types of notifications to enable independently. Finally you update the user interface ❸ with the channels connection status and URI, if one exists.

When the channel is first created, the MPN channel URI doesn't exist and the channel URI is retrieved asynchronously from the MPN service. You want to display the URI once it's set, so you subscribe to the `HttpNotificationChannel`'s `ChannelUriUpdated` event. The newly assigned URI is read from the `ChannelUri` property of the event's `NotificationChannelUriEventArgs` instance:

```
void channel_ChannelUriUpdated(object sender,
    NotificationChannelUriEventArgs e)
{
   Dispatcher.BeginInvoke(() => channelUri.Text = e.ChannelUri.ToString());
}
```

The channel class also exposes events named `ErrorOccurred` and `ConnectionStatusChanged`, which might be useful for troubleshooting notification connections. We leave it to you to implement the `ErrorOccurred` and `ConnectionStatusChanged` event handlers. Notification channels remain opened even when the application terminates and it's important to call `HttpNotificationChannel`'s `Close` method when the application no longer wants to receive notification messages.

9.2.5 Looking for navigation parameters

When the application is launched from a toast notification, query string parameters will be sent to the application. To fully demonstrate toast notifications, you should display the navigation URI to the user when the sample application is launched. The best place to obtain the navigation URI and update the user interface is inside the `OnNavigatedTo` method:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    notificationMessage.Text =
        string.Format("Launched with Uri:\n{0}", e.Uri);
}
```

The navigation URI is read from the `Uri` property of the `NavigationEventArgs` and is written to the `notificationMessage` `TextBlock`. You also use the `notificationMessage` to display information from toast notifications received while the client application is running.

9.2.6 *In-app notifications*

When toasts are sent while the application is running, the toasts won't appear in the normal spot in the UI. In fact they won't appear at all, unless the application explicitly handles the notification. Raw notifications will also be completely lost unless the application is running and listening.

A running application can receive toast and raw notifications by subscribing to `ShellToastNotificationReceived` and `HttpNotificationReceived` respectively. We leave listening for raw notifications with the `HttpNotificationReceived` event as an exercise for the reader. To see `ShellToastNotificationReceived` events in action, subscribe to it in the `SetupChannel` method:

```
channel.ShellToastNotificationReceived +=
    channel_ShellToastNotificationReceived;
```

The payload of a toast notification contains three strings containing the toast's title, content, and navigation URI. These strings are read from the `Collection` property of the `NotificationEventArgs` instance passed to the `ShellToastNotificationReceived` event handler. The next listing shows how your sample application reads the notification data and updates the user interface.

Listing 9.5 Receiving toast notifications

```
void channel_ShellToastNotificationReceived(object sender,
    NotificationEventArgs e)
{
    string title, content, parameter;

    e.Collection.TryGetValue("wp:Text1", out title);
    e.Collection.TryGetValue("wp:Text2", out content);
    e.Collection.TryGetValue("wp:Param", out parameter);

    string message = string.Format("Toast notification received.\nTitle:"
        + " {0}\nContent: {1}\nParameter: {2}\n\n{3}",
        title, content, parameter, DateTime.Now);

    Dispatcher.BeginInvoke(() => notificationMessage.Text = message);
}
```

① **Read notification payload**

The `Collection` property is a dictionary of string objects and you read payload data using the `TryGetValue` method ①. The dictionary will only contain entries if the sending application provided data when it sent the notification. The key names are defined by an XML schema governing notifications. You'll see an example of the XML used to create notifications later in the chapter as you implement the NotificationSimulator.

You need to implement one last feature in the client application before moving back to the simulator. You need to share the client's channel URI with the simulator application. The easiest way to implement sharing is by copying the channel URI to the clipboard.

9.2.7 Copying the channel URI

In a normal push notification solution, the client would share the channel URI by sending it to a web service as part of a web request. Both the NotificationClient and the NotificationSimulator run on the Windows Phone, and the operating system prevents applications from sharing data, with the exception of sharing text via the clipboard. You learned how to use the clipboard in chapter 2. You start by adding an application bar button the user can tap to initiate the copy:

```
<shell:ApplicationBarIconButton Text="copy" Click="copy_Click"
    IconUri="/Images/appbar.save.rest.png" />
```

In the button's `Click` event handler, you use the static `SetText` method of the `Clipboard` class to place the channel's URI on the clipboard:

```
private void copy_Click(object sender, EventArgs e)
{
    Clipboard.SetText(channelUri.Text);
}
```

Now that the channel URI can be placed on the clipboard, you need to update the NotificationSimulator application so that the user can paste the channel URI into the simulator. Open up MainPage.xaml in the NotificationSimulator project and add a new `PivotItem` with a header value of `channel`:

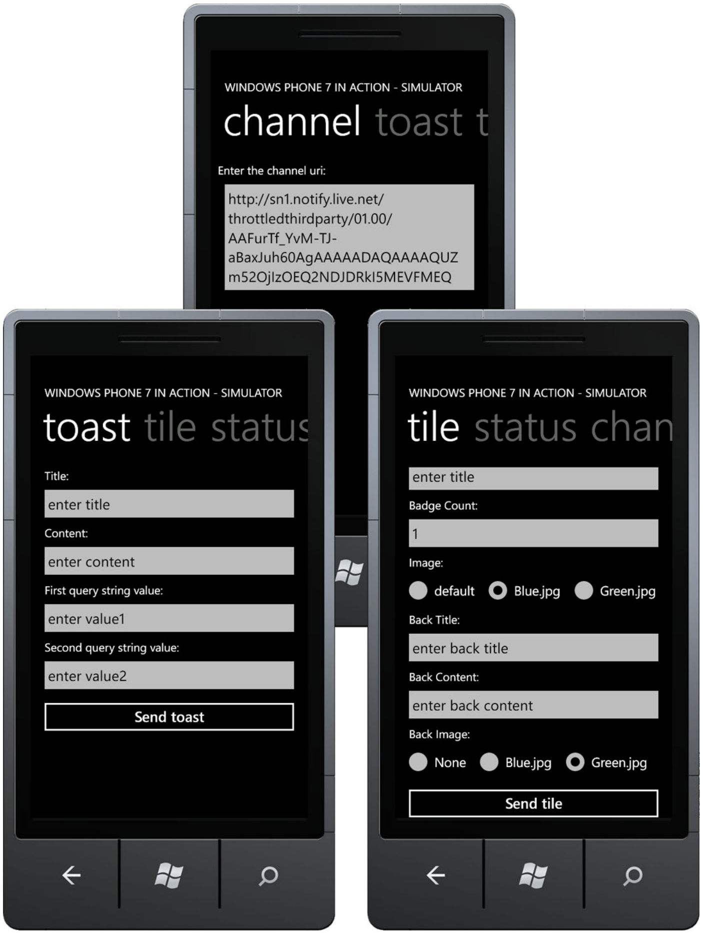
```
<controls:PivotItem Header="channel">
    <StackPanel>
        <TextBlock Text="Enter the channel uri:" />
        <TextBox x:Name="channelUri" TextWrapping="Wrap" />
    </StackPanel>
</controls:PivotItem>
```

The `PivotItem` contains a `TextBlock` for a label, and a `TextBox` to hold the pasted-in channel URI. You don't need to add any other code, since the SIP keyboard displayed when the `TextBox` has the focus provides its own Paste button. With the channel URI pasted into the simulator, you're ready to start pushing notifications.

9.3 Simulating a push notification service

Real solutions would have a web service built using ASP.NET, Node.js, or some other server-side technology. This book isn't about server-side technologies, so you're taking a shortcut and building a phone application that sends notifications to the MPN service. The code we show you for performing the web request is nearly identical to the code you might implement in an ASP.NET server-side implementation. Screenshots of the simulator application's user interface are shown in figure 9.6.

You'll update the NotificationSimulator project to include new screens to allow you to enter information for toast or tile notifications and tap a button to send the notification. The `Click` event handler code will call through to a new class, named `NotificationService`, which will issue a HTTP POST request to the MPN service.



WINDOWS PHONE 7 IN ACTION - SIMULATOR

channel toast t

Enter the channel uri:

http://sn1.notify.live.net/throttledthirdparty/01.00/AAFurTf_YvM-TJ-aBaxJuh60AgAAAAADAQAAAAQUZm52OjIzOEQ2NDJDRkI5MEVFMEQ

WINDOWS PHONE 7 IN ACTION - SIMULATOR

toast tile status

Title: enter title

Content: enter content

First query string value: enter value1

Second query string value: enter value2

WINDOWS PHONE 7 IN ACTION - SIMULATOR

tile status chan

enter title

Badge Count: 1

Image: ○ default ● Blue.jpg ○ Green.jpg

Back Title: enter back title

Back Content: enter back content

Back Image: ○ None ○ Blue.jpg ● Green.jpg

Figure 9.6 Screenshots of the NotificationSimulator

9.3.1 *Issuing HTTP web requests*

Windows Phone provides several ways to interact with web services, including those that expose SOAP and REST APIs. SOAP and REST web services have different design and structures, but both communicate using the HTTP protocol, the same protocol used by traditional HTML web servers.

> **NOTE** One way to build a client for SOAP-based web services is with the WCF tools and libraries. An example application using WCF can be found in the Bing Maps sample in chapter 13.

In this section, we introduce you to `HttpWebRequest`, a `System.Net` class you can use to communicate with HTTP servers such as the Microsoft Push Notification Service. The `HttpWebRequest` class isn't unique to Windows Phone, and exists in both Silverlight and the desktop .NET Framework. `HttpWebRequest` allows you to control how the HTTP request is constructed and sent. For example, you can select the HTTP verb, set the time-out, add header values or cookies to the request, and manage credentials and certificates. Covering the full breadth and depth of the `HttpWebRequest` API is beyond the scope of this book, but you'll use it to POST notifications to the MPN.

There are three steps in sending a web request using the `HttpWebRequest` class. The first step is to create the request and set the HTTP verb, URI, and headers. The second step entails writing the body of the request. The final step is to read the web server's response. The `NotificationService` class starts the process of sending a request in a method named `Post`, shown in the next listing.

Listing 9.6 Posting a notification using `HttpWebRequest`

```
private void Post(Uri channel, string payload, string target,
    string interval)
{
    HttpWebRequest request =
        (HttpWebRequest)HttpWebRequest.Create(channel);
    request.Headers["X-NotificationClass"] = interval;
    request.Headers["X-MessageID"] = Guid.NewGuid().ToString();
    if (target.Length > 0)
        request.Headers["X-WindowsPhone-Target"] = target;
    request.Method = "POST";
    request.ContentType = "text/xml; charset=utf-8";

    request.BeginGetRequestStream(WriteCallback,
        new RequestStreamState { Request = request, Payload = payload });
}
```

① **Create custom headers**

② **Asynchronously write body of request**

You start by creating a new `HttpWebRequest` using the static `Create` method. Next you add custom web headers ① defined by the MPN service. The header named `X-NotificationClass` specifies how quickly the notification should be sent and is specific to each type of notification. The `X-MessageID` header is optional, and is provided to allow the sending application to match web requests with web responses. The type of notification is identified with the header named `X-WindowsPhone-Target`. The target header is left blank for raw notifications, set to `toast` for toast, and set to `token` for tile notifications. After setting the HTTP verb, you start the second step in the request process by calling `BeginGetRequestStream` ②.

`BeginGetRequestStream` is an asynchronous method that takes a callback method and a user-defined state object. When the web request is ready for the body to be written, the callback method will execute and the user-defined state object will be passed in. Here's a custom state class named `RequestStreamState`:

```
class RequestStreamState
{
    public HttpWebRequest Request;
    public string Payload;
}
```

`RequestStreamState` allows you to pair the payload and the request object together so that they're both available to the `WriteCallback` method. The following listing shows the `WriteCallback` implementation.

Listing 9.7 Writing the payload to the request stream

```
void WriteCallback(IAsyncResult result)
{
    RequestStreamState state = (RequestStreamState)result.AsyncState;
    using (var stream = (Stream)state.Request.EndGetRequestStream(result))
    {
        byte[] messageBytes =
            Encoding.UTF8.GetBytes(state.Payload);
        stream.Write(messageBytes, 0, messageBytes.Length);
    }
    state.Request.BeginGetResponse(
        ReadCallback, state.Request);
}
```

❶ **Convert payload to bytes**

❷ **Asynchronously read response**

The first thing you do in `WriteCallback` is pull the `RequestStreamState` out of the result. Next you get the `Stream` associated with the request by calling `EndGetRequestStream`. Before you write the payload to the stream, you convert the string to a `byte` array ❶ using the `GetBytes` method of the `Encoding` class. The last step is to call `BeginGetResponse` ❷.

Following the .NET asynchronous pattern, `BeginGetResponse` takes a callback method and a user-defined state. The callback method is executed once the HTTP server has received your request and sent a response document. Your callback method is named `ReadCallback`, which is shown in the following listing, and you pass the original `HttpWebRequest` instance as the state object.

Listing 9.8 Receiving the response

```
void ReadCallback(IAsyncResult result)
{
    string message;
    HttpWebRequest request = (HttpWebRequest)result.AsyncState;
    try
    {
        HttpWebResponse response =
            (HttpWebResponse)request.EndGetResponse(result);

        message = string.Format(
            "Push request compeleted with:\n  {0}\n  {1}\n  {2}",
            response.Headers["X-NotificationStatus"],
            response.Headers["X-SubscriptionStatus"],
            response.Headers["X-DeviceConnectionStatus"]);
    }
    catch (Exception ex)
    {
        message = string.Format("{0} pushing notification: {1}",
            ex.GetType().Name, ex.Message);
    }
    Deployment.Current.Dispatcher.BeginInvoke(() =>
        MessageBox.Show(message));
}
```

❶ **Look at response headers**

❷ **Display response message**

The listing starts by casting the `AsyncState` property to `HttpWebRequest`. Next you call `EndGetResponse` to get the `HttpWebResponse` instance that represents the response document sent from the server. You read a few custom headers defined by the MPN service and format a message with them ❶. When an exception occurs while reading the response, you format an error message. Finally you display the message ❷ to the user using the `MessageBox` class.

`HttpWebReponse` provides several other properties and methods not covered by your sample application. The body content of the response can be read using `GetResponseStream`. The HTTP status code and description are returned by the `StatusCode` and `StatusDescription` properties.

Now that the `NotificationService` class is ready to push notifications, you'll try sending a toast notification.

9.3.2 Sending toast notifications

The payload for a toast notification is a string containing a well-defined XML document. The XML document uses the WPNotification namespace (xmlns). The root element is `Notification`, with a single `Toast` element:

```
<?xml version="1.0" encoding="utf-8"?>
<wp:Notification xmlns:wp="WPNotification">
    <wp:Toast>
        <wp:Text1>your title</wp:Text1>
        <wp:Text2>your content</wp:Text2>
        <wp:Param>your parameter</wp:Param>
    </wp:Toast>
</wp:Notification>
```

The `Toast` element should contain three child elements named `Text1`, `Text2`, and `Param`. `Text1` is the toast's title, whereas the toast's content is placed in `Text2`. `Param` contains the launch URI used when the user taps a displayed toast notification.

> **NOTE** The Push Notification Service isn't the only way to display a toast notification to the user. A background agent can display a toast notification to the user using the `Show` method of the `ShellToast` class.

In your sample applications, you'll send a toast with a launch URI containing two query string parameters. Add a new `PivotItem` to the simulator's MainPage.xaml to allow the user of the simulator to enter the title, content, and query string parameters for the toast. The markup for the new `PivotItem` is shown in the next listing.

Listing 9.9 Markup for entering a toast notification into the simulator

```
<controls:PivotItem Header="toast">
    <StackPanel>
        <TextBlock Text="Title:"  />
        <TextBox x:Name="toastTitle" Text="enter title" />
        <TextBlock Text="Content:"  />
```

❶ **Label pivot item**

```
        <TextBox x:Name="toastContent" Text="enter content" />
        <TextBlock Text="First query string value:"  />
        <TextBox x:Name="toastValue1" Text="enter value1" />
        <TextBlock Text="Second query string value:"  />
        <TextBox x:Name="toastvalue2" Text="enter value2" />
        <Button Content="Send toast" Click="sendToast_Click" />
    </StackPanel>
</controls:PivotItem>
```

❷ **Button to send toast**

The new `PivotItem` is added and is given the label toast ❶. Four pairs of `TextBlocks` and `TextBoxes` are stacked in the user interface. Each pair is a label and input control to capture the values to be sent in the toast. Finally, a `Button` ❷ is provided which the user taps to send the toast notification. The `Click` event handler, named `sendToast_Click`, calls a new `NotificationService` method named `SendToast`:

```
private void sendToast_Click(object sender, RoutedEventArgs e)
{
    Uri channel;
    if (Uri.TryCreate(channelUri.Text, UriKind.Absolute, out channel))
    {
        service.SendToast(channel, toastTitle.Text, toastContent.Text,
            string.Format("/MainPage.xaml?value1={0}&amp;value2={1}",
                toastValue1.Text, toastvalue2.Text));
    }
}
```

Inside `sendToast_Click`, the pasted channel URI is read from the `channelUri TextBox`. The launch URI is constructed by concatenating the two query string values.

The new `SendToast` method formats the payload and calls the `Post` method:

```
public void SendToast(Uri channel, string title, string content,
    string launchUri)
{
   string payload = string.Format(ToastPayload, title, content, launchUri);
   Post(channel, payload, "toast", "2");
}
```

The payload XML string is constructed by using the `Format` method to replace placeholders in the constant `ToastPayload` string with the specified title, content, and launch URI. The `ToastPayload` constant, not shown here for space, contains the XML for a toast notification, with placeholders in the appropriate places. The payload is passed to the `Post` method along with the channel URI, an `X-WindowsPhone-Target` string of `toast`, and an interval of 2 for the `X-NotificationClass` value.

Valid interval values for toast notifications are 2, 12, or 22. A value of 2 indicates that the MPN service should forward the notification immediately, whereas values of 12 and 22 indicate wait times of 450 seconds and 900 seconds respectively.

9.3.3 Using notifications to update a tile

The payload for a tile notification is also a string containing a well-defined XML document. The XML document uses the `WPNotification` namespace with a `Notification` root element. Inside the `Notification` element is a single `Tile` element. The tile notification XML is shown in the next listing.

Listing 9.10 Payload xml format for a tile notification

```
<?xml version="1.0" encoding="utf-8"?>
<wp:Notification xmlns:wp="WPNotification">
    <wp:Tile ID="tile uri">
        <wp:BackgroundImage>your image</wp:BackgroundImage>
        <wp:Count Action="Clear">your badge count</wp:Count>
        <wp:Title Action="Clear">your title</wp:Title>
        <wp:BackBackgroundImage Action="Clear">your back image
            </wp:BackBackgroundImage>
        <wp:BackTitle Action="Clear">your back title</wp:BackTitle>
        <wp:BackContent Action="Clear">your back content</wp:BackContent>
    </wp:Tile>
</wp:Notification>
```

① **Optional ID attribute**

② **Optional Action attribute**

The tile XML format includes each of the six tile properties. `BackgroundImage`, `Count`, and `Title` update the front of the tile, whereas `BackBackgroundImage`, `BackTitle`, and `BackContent` update the back of the tile. The `Notification` element defines an optional ID attribute ① that can be used to update a secondary tile instead of the main application tile. Five of the six tile properties define an optional `Action` attribute. When the `Action` attribute is set to `Clear` ②, the corresponding tile property is set back to its default value.

You'll update the simulator's user interface with a new `PivotItem` for setting each of the six tile notification properties. The user interface for containing controls for each of the six properties doesn't nicely fit into a single screen. The tile notification `PivotItem`, shown in the next listing, uses a `ScrollViewer` that allows the user to scroll the form to see off-screen input controls.

Listing 9.11 XAML markup for sending a tile notification from the simulator

```
<controls:PivotItem Header="tile">
    <ScrollViewer>
        <StackPanel>
            <TextBlock Text="Title:"/>
            <TextBox x:Name="tileTitle" Text="enter title"/>
            <TextBlock Text="Badge Count:"/>
            <TextBox x:Name="tileBadgeCount" Text="1"
                InputScope="Number"/>
            <TextBlock Text="Image:"/>
            <StackPanel Orientation="Horizontal">
                <RadioButton x:Name="tileDefaultImage" GroupName="tile"
                    Content="default"/>
                <RadioButton x:Name="tileBlueImage" GroupName="tile"
                    IsChecked="True" Content="Blue.jpg"/>
                <RadioButton x:Name="tileGreenImage" GroupName="tile"
                    Content="Green.jpg"/>
            </StackPanel>
```

① **Use number keyboard**

Pick background image ②

```
            <TextBlock Text="Back Title:"/>
            <TextBox x:Name="tileBackTitle" Text="enter back title"/>
            <TextBlock Text="Back Content:"/>
            <TextBox x:Name="tileBackContent" Text="enter back content"/>
            <TextBlock Text="Back Image:"/>
            <StackPanel Orientation="Horizontal">
                <RadioButton x:Name="tileBackNoImage" GroupName="backTile"
                    Content="None"/>
               <RadioButton x:Name="tileBackBlueImage" GroupName="backTile"
                    Content="Blue.jpg"/>
                <RadioButton x:Name="tileBackGreenImage" IsChecked="True"
                    GroupName="backTile" Content="Green.jpg"/>
            </StackPanel>

            <Button Content="Send tile" Click="sendTile_Click"/>
        </StackPanel>
    </ScrollViewer>
</controls:PivotItem>
```

You start by declaring a new `PivotItem` that contains a `StackPanel` inside a `ScrollViewer`. Most of the user interface is built from pairs of `TextBlocks` and `TextBoxes` allowing the user to enter the titles, count, and content values to be sent with the tile notification. The `InputScope` of the `TextBox` for entering the count value is set to `Number` ①, to make it easier for the user. There are two groups of `RadioButtons` ② on the form for picking which image file should be used for the background images.

The last item added to the tile `PivotItem` is a `Button` the user taps to send the notification. The following listing details the implementation of the button's `Click` handler. After reading and translating values from the user interface, the `NotificationService` method named `SendTile` is called.

Listing 9.12 The `SendTile` click handler

```
private void sendTile_Click(object sender, RoutedEventArgs e)
{
    Uri channel;
    if (Uri.TryCreate(channelUri.Text, UriKind.Absolute, out channel))
    {
        string imagePath = tileDefaultImage.IsChecked.Value ?
            "Background.png" : tileBlueImage.IsChecked.Value
                ? "Blue.jpg" : "Green.jpg";
        string backImagePath = tileBackNoImage.IsChecked.Value ? "" :
            tileBackBlueImage.IsChecked.Value ? "Blue.jpg" : "Green.jpg";

        int badgeCount;
        Int32.TryParse(tileBadgeCount.Text, out badgeCount);

        service.SendTile(channel, imagePath, badgeCount, tileTitle.Text,
            backImagePath, tileBackTitle.Text, tileBackContent.Text);
    }
}
```

Convert count text to integer ①

First the image `RadioButtons` are examined to determine which image should be sent in the notification. The NotificationClient application includes Background.png, Blue.jpg, and Green.jpg files in the .xap file. You parse the value entered into the `tileBadgeCount TextBox` ❶. Finally you send all six property values to `SendTile`.

The implementation of `SendTile`, shown in the next listing, isn't as straightforward as the implementation of `SendToast`.

Listing 9.13 Building the tile payload

```
const string Clear = "Action\"Clear\"";            ◁—— ❶ Text for Action attributes

public void SendTile(Uri channel, string tileId, string imagePath,
    int badgeCount, string title, string backImagePath,
    string backTitle, string content)
{
    string badgeCountAction = "", titleAction = "",
        backImagePathAction = "", backTitleAction = "", contentAction = "";

    if (badgeCount < 1) badgeCountAction = Clear;
    if (string.IsNullOrEmpty(title)) titleAction = Clear;
    if (string.IsNullOrEmpty(backImagePath)) backImagePathAction = Clear;
    if (string.IsNullOrEmpty(backTitle)) backTitleAction = Clear;
    if (string.IsNullOrEmpty(content)) contentAction = Clear;

    string payload = string.Format(TilePayload, imagePath,
        badgeCountAction, badgeCount, titleAction, title,
        backImagePathAction, backImagePath, backTitleAction, backTitle,
        contentAction, content);

    Post(channel, payload, "token", "1");
}
```

`SendTile` examines each of the properties to see whether it contains a valid value, and if not, sets the `Action` attribute of related XML element to the value `Clear`. You define a constant string ❶ to use when formatting the payload XML string. The payload XML string is constructed by using the `Format` method to replace placeholders in the constant `TilePayload` string with the specified title, content, and launch URI. The `TilePayload` constant, not shown here for space, contains the XML for a tile notification, with placeholders in the appropriate places. The payload is passed to the `Post` method along with the channel URI, an `X-WindowsPhone-Target` string of `token`, and an interval of 1 for the `X-NotificationClass` value.

Valid interval values for tile notifications are 1, 11, or 21. A value of 1 indicates that the MPN service should forward the notification immediately, whereas values of 11 and 21 indicate wait times of 450 seconds and 900 seconds respectively.

9.4 Tiles without all the pushiness

The Windows Phone 7.1 SDK allows an application to update a tile without using tile notifications. The `ShellTile` class in the `Microsoft.Phone.Shell` namespace exposes tiles to application code. You first saw how to use the `ShellTile` class in the Hello World example, when you pinned a secondary tile to the start screen.

An application adds a new tile to the start screen by creating a `StandardTileData` instance and then calling the static `ShellTile.Create` method. An application can then determine whether any of its tiles are currently pinned to the start screen using the static `ActiveTiles` collection of the `ShellTile` class. `ShellTile` and `StandardTileData` can also be used to locally update a tile.

You'll add some code to the NotificationClient sample application to demonstrate how to update a tile without the Push Notification service. You start by copying the user interface code in listing 9.11 from the NotificationSimulator into the NotificationClient. Update the send `Button` to call a `Click` event handler named `updateTile_Click`. The event handler is shown in the next listing.

Listing 9.14 Updating a tile

```
private void updateTile_Click(object sender, RoutedEventArgs e)
{
    string imagePath = tileDefaultImage.IsChecked.Value ?
        "Background.png" : tileBlueImage.IsChecked.Value ?
            "Blue.jpg" : "Green.jpg";
    string backImagePath = tileBackNoImage.IsChecked.Value ? "" :
        tileBackBlueImage.IsChecked.Value ? "Blue.jpg" : "Green.jpg";
    int badgeCount;
    Int32.TryParse(tileBadgeCount.Text, out badgeCount);

    ShellTile tile = ShellTile.ActiveTiles.First();
    StandardTileData tileData = new StandardTileData
    {
        BackgroundImage = new Uri(imagePath, UriKind.Relative),
        Count = badgeCount,
        Title = tileTitle.Text,
        BackBackgroundImage =
           new Uri(backImagePath, UriKind.Relative),
        BackTitle = tileBackTitle.Text,
        BackContent = tileBackContent.Text,
    };
    tile.Update(tileData);
}
```

① **First tile is primary tile**

② **Update tile properties**

The initial part of the listing is similar to that `sendTile_Click` method in the NotificationSimulator. The image filenames are set based on the checked `RadioButtons`, and the count is parsed from user input. The first tile in the `ActiveTiles` collection is always the primary application tile, even if the tile isn't pinned to the start screen, so you use the `First` ① extension method to retrieve the primary tile. Next, you create an instance of `StandardTileData` and set the six properties ② of the tile. Finally you call the `Update` method to change the tile.

Now you're ready to send some push notifications. You start by pinning the NotificationClient to the start screen. Next you run the NotificationClient sample application. Once the channel URI is displayed, tap the application bar button to copy it to the clipboard. Now run the NotificationSimulator application and paste the channel URI into the `TextBox` you put in the channel `PivotItem`. Now you can use the tile and toast pivots to send notifications.

Push notifications provide a mechanism where a web service can indirectly communicate with a Windows Phone application, whether or not the application is running. Indirect communication isn't sufficient for many applications that require direct connections between the client and the web service. Some applications can use the `HttpWebRequest` class for direct connections initiated by the client, but it doesn't support server-initiated interactions. Applications whose network interaction requirements can't be satisfied by push notification or `HttpWebRequest` can use sockets to enable communication between client and server.

9.5 Communicating with sockets

The Windows Phone 7.1 SDK implements the networking APIs for communicating via sockets to enable scenarios such as instant messaging, multiplayer games, and integration with non-HTTP services. Socket support has existed in the desktop .NET Framework and Silverlight for some time, and wasn't implemented in the first release of Windows Phone 7. The Windows Phone 7.1 SDK supports both TCP and UDP protocols.

TCP stands for *Transmission Control Protocol* and is widely used on the network and in applications that require guaranteed delivery of network transmissions. TCP is a connection-oriented protocol and always makes sure that data sent by one endpoint will be received by the other endpoint.

UDP stands for *User Datagram Protocol*. UDP has no error checking or verification after transferring data so there's no guarantee that the data will arrive at its destination. The advantage of UDP is that it's very fast and so it's used in real-time applications that don't have any critical data transfers. The UDP support in the Windows Phone 7.1 SDK includes both unicast and multicast clients. Table 9.3 compares TCP and UDP.

Table 9.3 TCP versus UDP

TCP	UDP
Works on data streams (not on data blocks)	Works on data blocks up to 64 kilobytes in size
Continuous connection	Connection-less
Guaranteed delivery	Doesn't guarantee data will arrive at the remote endpoint

Unicast is a one-to-one communication between client and server and all data packets are sent from a single source to a destination endpoint. *Multicast* is used for sending a data packet from a source to multiple destination endpoints.

If you've used sockets with Silverlight or the desktop .NET Framework, you're probably already familiar with the topics we're going to cover in the rest of this chapter. We're going to show you how to use TCP sockets by building a simple client/server chat application.

9.6 Implementing a chat application with TCP sockets

You're going to use TCP sockets to implement a sample chat application that connects to a chat server and sends simple text messages to all connected chat clients. The sample application will demonstrate how to open a socket connection. Once opened, the application will asynchronously send and receive chat messages.

> **NOTE** In this book, we're going to focus only on the client-side socket programming and won't cover the server-side code. The server project is included in the sample code for the book. The server code requires either C# Express or Visual Studio Professional.

The data stream sent in between the client and server is formatted as an array of bytes. The array can contain any data in any format, as long as both the client and server agree on the contents of the data stream. In your sample application, the data stream will contain three strings concatenated together with a semicolon. The three parts of the data payload are the command, the user's name, and the chat message:

```
command;username;message
```

The first part of the payload is a command that identifies the type of message being sent from a client. Your sample application supports two commands—`CONNECT` and `CHAT`. A client registers with the chat server by sending the `CONNECT` command. A client sends the `CHAT` command when a new text message should be shared with all connected clients. The second part of the payload contains the username entered into the chat client. The last part of the payload is the text of the chat message entered by a user.

When the Chit-chat server starts, it'll begin listening on port 22222 for new socket connections. The client will use the server's IP address and port to create a socket connection and send a `CONNECT` command to let any other connected clients know that a new user joined the chat. Once connected, the client will start listening for communication from the server. When the client sends a `CHAT` command, the server will broadcast it to all connected clients. Connected clients will receive the message and will update their user interface.

9.6.1 Building the Chit-chat client

You're going to create a new project for the chat client sample application. Use the Windows Phone Application project template and name the project ChitChatClient. Open up MainPage.xaml so that you can add the markup for the user interface. A screenshot of the finished application is shown in figure 9.7.

The user interface contains input controls for the user's name and the server's IP address. A list of chat messages is displayed on the screen. A third input control allows the user to enter new messages. The `ApplicationBar` contains buttons for connecting to the server and sending new chat messages. The next listing shows the XAML markup for the user interface.

Listing 9.15 Defining the chat client user interface

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <StackPanel x:Name="connectPanel">
        <TextBlock Text="User name" />
        <TextBox x:Name="userName" />
        <TextBlock Text="Server IP address" />
        <TextBox x:Name="serverIpAddress" InputScope="Number" />
    </StackPanel>
    <ListBox x:Name="messageList" Grid.Row="1" />
    <TextBox x:Name="messageText" Grid.Row="2" Margin="0,0,0,25" />
</Grid>
```

① **Stack login controls**

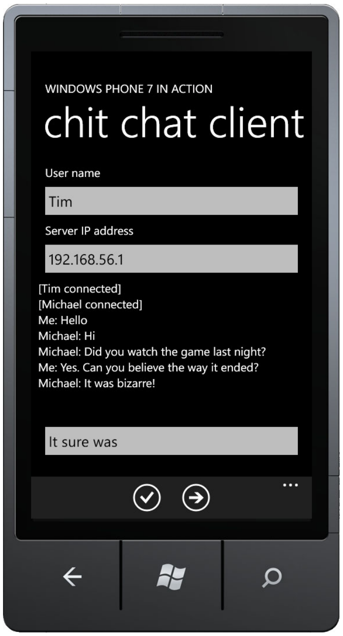
② **Messages display in list**

You start by splitting the `ContentPanel` grid into three rows, allowing the second row to take up any extra space. Next you add a `TextBox` for the user's name and a second `TextBox` for the server's IP address, using `TextBlocks` for labels. All four login controls are placed into a `StackPanel` ①. When new messages arrive, they'll be added to a collection of strings and displayed in a `ListBox` ②. Finally, you add a third `TextBox` for new chat messages.

Add a class-level field to hold the collection of messages sent from the server. The collection is bound to the `ListBox` in the `MainPage` constructor:

```
readonly ObservableCollection<string> messages =
    new ObservableCollection<string>();

public MainPage()
{
    InitializeComponent();
    messageList.ItemsSource = messages;
}
```



WINDOWS PHONE 7 IN ACTION

chit chat client

User name

Tim

Server IP address

192.168.56.1

[Tim connected]
[Michael connected]
Me: Hello
Michael: Hi
Michael: Did you watch the game last night?
Me: Yes. Can you believe the way it ended?
Michael: It was bizarre!

It sure was

Figure 9.7 A screenshot of the Chit-Chat client

The user connects to the chat server by tapping a button in the application bar. Chat messages are sent using a second button. Add the markup for the application bar buttons:

```
<shell:ApplicationBarIconButton Text="connect" Click="connect_Click"
    IconUri="/Images/appbar.check.rest.png" />
<shell:ApplicationBarIconButton Text="send" Click="send_Click"
    IconUri="/Images/appbar.next.rest.png"  />
```

The sample application uses two images from the Windows Phone 7.1 SDK. Add the specified images to a new project folder named Images.

9.6.2 Connecting to the server

When the user connects to the server, a three-step process is initiated. First the socket is created and then a connection is established asynchronously. Once the connection is established, the application begins listening for messages from the server. The `Click` event handler for the connect button, shown in the next listing, performs the first step.

Listing 9.16 Implementing the `click` handler

```
const int PORT = 22222;
IPEndPoint endpoint;
Socket socket;
string user;

private void connect_Click(object sender, EventArgs e)
{
    Disconnect();
    IPAddress address;
    if (IPAddress.TryParse(serverIpAddress.Text, out address))
    {
        endpoint = new IPEndPoint(address, PORT);
        socket = new Socket(AddressFamily.InterNetwork,
            SocketType.Stream, ProtocolType.Tcp);
        user = userName.Text;
        Connect();
    }
}
```

Declare class-level fields and constants

① **Parse server's IP address**

② **Create socket**

Before you create a new connection, you disconnect any existing connection. Next you parse the text entered into the `serverIpAddress` `TextBox` ①. If the text is successfully parsed, you create a new `IPEndPoint` instance with the specified IP address and port number 22222. You also create a new `Socket` instance ② using the `Stream` `SocketType` and `Tcp` `ProtocolType`, which identify the `Socket` as a client/server TCP connection. The `InterNetwork` `AddressFamily` value tells the socket to use the IP version 4 addressing scheme. Finally you store the entered username and call the `Connect` method.

Before looking at the implementation of the `Connect` method, let's take a quick look at the `Disconnect` method. `Disconnect` closes and cleans up an existing connection calling the `Socket`'s `Close` method and setting the `endpoint` and `socket` fields to `null`:

```
private void Disconnect()
{
    endpoint = null;
    if (socket != null)
    {
        socket.Close();
        socket = null;
    }
}
```

The `Connect` method is where you establish a connection to the server. As part of the initial connection, you send a `CONNECT` command as the data payload of the socket message. Before calling the `ConnectAsync` method of the `Socket` class, you need to create an instance of `SocketAsyncEventArgs`. The `SocketAsyncEventArgs` class is used to specify the endpoint address, the payload, and the `Completed` event handler. You pass the `SocketAsyncEventArgs` instance as a parameter to the `ConnectAsync` method of `Socket`:

```
private void Connect()
{
    byte[] buffer = FormatPayload (CONNECT, string.Empty);
    var connectArgs = new SocketAsyncEventArgs{RemoteEndPoint = endpoint};
    connectArgs.Completed += connect_Completed;
    connectArgs.SetBuffer(buffer, 0, buffer.Length);
    socket.ConnectAsync(connectArgs);
}
```

The payload is constructed by concatenating the command, username, and chat text into a single string. The combined string is transformed into an array of bytes using the `Encoding` class:

```
private byte[] CreateMessage(string command, string text)
{
    string message = string.Format("{0};{1};{2}", command, user, text);
    return Encoding.UTF8.GetBytes(message);
}
```

Once the connection is established, the `Completed` event is raised, and your `connect_Completed` event handler is called. The following listing shows the implementation of the event handler, where you check whether an error occurred while connecting to the server. If the connection was successful, you start listening for messages from the server.

Listing 9.17 Listening for messages

```
const int MAX_BUFFER_SIZE = 1024;

void connect_Completed(object sender, SocketAsyncEventArgs e)
{
    e.Completed -= connect_Completed;
    if (e.SocketError != SocketError.Success)
    {
        Dispatcher.BeginInvoke(() =>
        {
            MessageBox.Show("Unable to connect to the chat server.",
                "Error", MessageBoxButton.OK);
            Disconnect();
        });
    }
    else
    {
        var receiveArgs = new SocketAsyncEventArgs();
```

① **Display errors to user**

```
        receiveArgs.RemoteEndPoint = endpoint;
        receiveArgs.SetBuffer(new byte[MAX_BUFFER_SIZE],0,MAX_BUFFER_SIZE);
        receiveArgs.Completed += receive_Completed;
        socket.ReceiveAsync(receiveArgs);
    }
}
```

② **Listen for messages from server**

The first thing you do in the listing is unsubscribe to the `Completed` event for the `SocketAsyncEventArgs` instance passed into the event handler. Next you look at the `SocketError` property to see whether an error occurred while connecting to the server. If an error occurred, you display a message to the user ① and close the socket. If the connection is successful, you immediately create a new instance of `SocketAsyncEventArgs`. This time, you create an empty array of 1024 bytes that will be used when the socket receives a message. After setting the endpoint address and a `Completed` event handler, you pass the new instance of `SocketAsyncEventArgs` to the socket's `ReceiveAsync` method ②.

9.6.3 Receiving messages from the server

When a message is received from the server, the `ReceiveAsync Completed` event handler is called. In your sample application, the event handler is named `received_Completed`, which is shown in the following listing. Inside the event handler you split apart the three strings in the payload and update the user interface.

Listing 9.18 Receiving a message

```
void receive_Completed(object sender, SocketAsyncEventArgs e)
{
    if (e.SocketError != SocketError.Success)
    {
        Dispatcher.BeginInvoke(() =>
            MessageBox.Show("An error occured during server communication."
                + e.SocketError));
    }
    else
    {
        string message = Encoding.UTF8.GetString(
            e.Buffer, 0, e.BytesTransferred);
        if (!string.IsNullOrEmpty(message))
        {
            string[] msgParts = message.Split(';');
            string newMessage;
            if (msgParts[0] == CONNECT)
                newMessage = string.Format("[{0} connected]", msgParts[1]);
            else
                newMessage = string.Format("{0}: {1}",
                    msgParts[1] == user ? "Me" : msgParts[1], msgParts[2]);
            Dispatcher.BeginInvoke(() => messages.Add(newMessage));
        }

        if (socket != null)
            socket.ReceiveAsync(e);
    }
}
```

① **Convert payload to string**

② **Split payload into three strings**

③ **Add new message to collection**

First you check whether an error occurred while waiting for a new message, and if so, you display the error to the user. Otherwise you convert the payload from the event args `Buffer` property and convert it into a `string` ❶ using the `Encoding` class. Next you split the payload into three strings ❷ containing the command, username, and chat message. You look at the command value to determine how to format the message. You add the formatted message to the collection that's bound to the `ListBox` in the user interface ❸. Finally you start listening for another message, reusing the `SocketAsyncEventArgs` instance.

9.6.4 Sending a message

Sending the message from Windows Phone is similar to the way that you connect and receive the data from server. You need to create an instance of `SocketAsyncEventArgs` and pass it to the `Socket`'s `SendAsync` method. In your sample application, you read the text entered in the `messageText` `TextBox` and send it to the server inside the `send_Click` event handler, shown in the next listing.

Listing 9.19 Sending a message

```
private void send_Click(object sender, EventArgs e)
{
    if (socket.Connected && !string.IsNullOrEmpty(messageText.Text))
    {
        byte[] buffer = FormatPayload(CHAT, messageText.Text);
        var sendArgs = new SocketAsyncEventArgs{RemoteEndPoint = endpoint};
        sendArgs.SetBuffer(buffer, 0, buffer.Length);
        sendArgs.Completed += send_Completed;
        socket.SendAsync(sendArgs);
    }
}
```

Validate chat message ❶

Before sending a `CHAT` command, you check whether the socket is still connected and whether the user has entered a message ❶. Next you call `FormatPayload` and create and initialize a new instance of `SocketAsyncEventArgs`. Finally you call the `SendAsync` method.

When the send operation has completed, `the send_Completed` event handler is called. You check whether an error occurred and display a message. Otherwise you clear out the text input box:

```
void send_Completed(object sender, SocketAsyncEventArgs e)
{
    e.Completed -= send_Completed;
    if (e.SocketError != SocketError.Success)
        MessageBox.Show("Your message could not be sent");
    else
        Dispatcher.BeginInvoke(() => messageText.Text = string.Empty);
}
```

You're now ready to test the ChitChatClient sample application. In order to test all real chat scenarios, you need two devices or emulators because you need to send the message back and forth between two different clients. Using C# express or Visual Studio Professional, open the ChitChatServer project from the sample source code and launch the application. Next run the ChitChatClient on two Windows Phones.

9.7 Summary

In this chapter you looked at three different methods to enable communication between a Windows Phone application and a remote service. Push notifications allow a remote service to send messages to the user of an application, even when the application isn't running on the phone. Push notifications are a one-way communication from server to client. One-way communications from client to server are possible with the `HttpWebRequest` class. A client uses `HttpWebRequest` to send a request to the server, and then reads the server's response. You used `HttpWebRequest` to send push notification request to the Microsoft Push Notification Service.

You also looked at how to use the `Socket` class to implement two-way client server communication. The `Socket` class allows you to develop applications that use TCP and UDP to communicate to one or more servers and endpoints.

In the next chapter we take a deep dive into the Silverlight controls built specifically for the Windows Phone. You'll learn how to work with the `ApplicationBar` and how to enable and disable button and menu items. We also take our first look at the `Panorama` control, which is used to power the built-in Music + Videos and Office Hubs. We wrap with a thorough examination of the `Pivot` control.

Part 3

Silverlight for Windows Phone

Even though Silverlight for Windows Phone is very similar to Silverlight for the browser, and by extension to Windows Presentation Foundation, there are new controls and concepts found only in the Windows Phone SDK. In Part 3, you'll learn how to use new Silverlight features to build applications that match the look and feel of Windows Phone.

While you've used the `ApplicationBar` and `Pivot` controls in sample applications throughout this book, chapter 10 takes a deep dive into these new controls. You'll also learn about the `Panorama` control, an essential ingredient for building hub-style user interfaces.

In chapter 11, we show you how to build applications that automatically adjust themselves when the phone is rotated from normal portrait mode into a landscape orientation. You'll also learn tricks to style common controls to match the Metro design, and how to control the software keyboard. Finally we introduce you to the Silverlight Toolkit for Windows Phone, a Codeplex project from Microsoft that includes additional user interface controls.

In chapters 12 and 13 you'll work with the MediaElement, Bing Map, and Web Browser controls. You'll also learn how to use the location service and Bing Map launchers to make a location-aware application.

10

ApplicationBar, Panorama, and Pivot controls

This chapter covers

- Working with the ApplicationBar
- Using Panoramas
- Pivoting views

The Windows Phone comes with a variation of Silverlight 4. This means that if you know how to build Silverlight applications, you know how to build Windows Phone applications. This chapter is the first of four chapters dedicated to Silverlight controls and how to use them on the phone platform. In this chapter we introduce controls that are unique to Silverlight for Windows Phone.

Windows Phone 7 has redefined how an application displays a toolbar and menu. Applications use the new `ApplicationBar` control to show up to four shortcut icons for the most common operations on the page. If additional options are available but don't fit in the bar containing shortcuts or a different level of granularity must be provided, developers can add textual menu items as well. In this chapter we'll cover how to create a basic menu which can be used in your applications.

We also introduce two new navigation controls: `Panorama` and `Pivot`. The `Panorama` control is used across the phone when the main screen of the application provides a rich graphical frontend to quickly access favorite and recently used content. Items appearing in the panorama are a linking to other pages where the content is viewed or manipulated. The `Pivot` control is similar to a tab control, where discrete pages display different sets of data or settings to the user. The email application is one example that uses a `Pivot` control to present separate pages for All, Unread, and Marked messages.

You've used the `ApplicationBar` and `Pivot` controls throughout the book, including several examples in earlier chapters in the book. In the previous chapters, your use of these controls was very basic. In this chapter we take a deeper look at `ApplicationBar` and `Pivot` controls, as well as introduce the `Panorama` control. We accomplish this through three sample applications. The first sample application demonstrates features of the application bar that we haven't discussed, such as disabling buttons and menu items, dynamically adding and removing items, and using different display modes. The second sample application focuses on how to build a hub-like application using the `Panorama` control. Working with the `Pivot` control is the theme of the final sample application, concentrating on how to efficiently pivot between views in an application.

10.1 Working with the ApplicationBar

The `ApplicationBar` is the new toolbar and menu paradigm created for the Windows Phone. It provides a toolbar for buttons with an expandable menu. The application bar is always placed at the bottom of the screen in portrait orientation, and on the side of the screen in landscape orientation (see figure 10.1). The application bar is

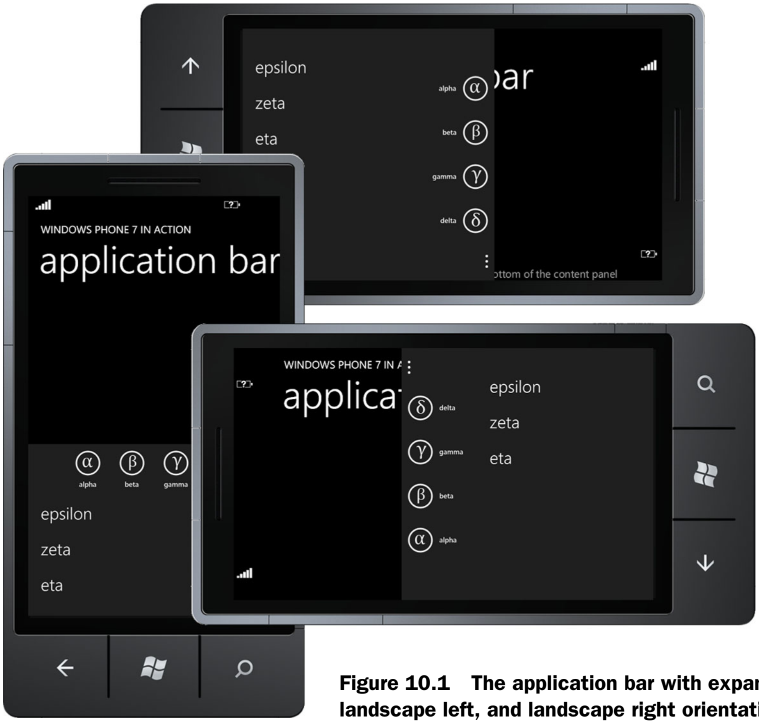

epsilon
zeta
eta
alpha
beta
gamma
delta
)ar
ottom of the content panel
WINDOWS PHONE 7 IN ACTION
application bar
alpha
beta
gamma
epsilon
zeta
eta
WINDOWS PHONE 7 IN A
applica
delta
gamma
beta
alpha
epsilon
zeta
eta

Figure 10.1 The application bar with expanded menu in portrait, landscape left, and landscape right orientations

associated with a page, and isn't global to the application. If you want the same features on multiple pages, you'll need to recreate the application bar items on every page.

The application bar is always positioned on the side of the screen where the start button lives. Touching the three dots at the edge of the application bar expands it to show the defined menu items. There are a maximum of 4 buttons and 50 menu items.

10.1.1 Building an application bar

We'll use a new sample application to demonstrate using an application bar in a Silverlight application. Create a new project, named ApplicationBar, using the Windows Phone Application template. Once the project is created, open up MainPage.xaml and add the markup declaring a new application bar. The following listing shows the XAML used to declare the application bar shown in figure 10.1.

Listing 10.1 Declaring application bar buttons and menu items

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar>
        <shell:ApplicationBar.MenuItems>
            <shell:ApplicationBarMenuItem Text="epsilon"
                Click="item_Clicked" />
            <shell:ApplicationBarMenuItem Text="zeta"
                Click="item_Clicked"/>
            <shell:ApplicationBarMenuItem Text="eta"
                Click="item_Clicked"/>
        </shell:ApplicationBar.MenuItems>

        <shell:ApplicationBarIconButton IconUri="/icons/alpha.png"
            Text="alpha" Click="item_Clicked" />
        <shell:ApplicationBarIconButton IconUri="/icons/beta.png"
            Text="beta" Click="item_Clicked" />
        <shell:ApplicationBarIconButton IconUri="/icons/gamma.png"
            Text="gamma" Click="item_Clicked"/>
        <shell:ApplicationBarIconButton IconUri="/icons/delta.png"
            Text="delta" Click="item_Clicked" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

① **Create application bar**

② **Add menu items to collection**

③ **Add buttons to content**

An application bar is created as a property of the `PhoneApplicationPage` using Silverlight's property element syntax ①. An application bar is represented by the `ApplicationBar` class, found in the `Microsoft.Phone.Shell` namespace, which is aliased in XAML as `shell`. The `ApplicationBar` contains two collection properties named `Buttons` and `MenuItems`. The `MenuItems` collection is populated by declaring `ApplicationBarMenuItem` objects inside of the `MenuItems` element ②. The `Buttons` collection is populated by declaring `ApplicationBarIconButton` objects as the content of the `ApplicationBar` ③.

The `ApplicationBarIconButton` class exposes `IconUri`, `Text`, and `IsEnabled` properties. `IconUri` must be provided and is a relative `Uri` to an image file that has been added to the project with a build action of `Content`. We'll discuss image files later in the chapter. The `Text` property is also required.

The `ApplicationBarMenuItem` class only has `Text` and `IsEnabled` properties. The `Text` property is required, and a value must be specified when the menu item is declared.

`ApplicationBarIconButton` and `ApplicationBarMenuItem` both provide a `Click` event that can be wired up to an event handler in XAML. The click events are basic event handlers and don't send a `RoutedEventArgs` or any other custom `EventArgs` type. If you need to know which button or menu item was clicked, you can use the event handler's sender parameter. In the preceding listing, you wired up the same click event handler to each of the buttons and menu items. The click event handler is implemented in the next listing.

Listing 10.2 Click event handler for application bar items

```
private void item_Clicked(object sender, EventArgs e)
{
    var button = sender as ApplicationBarIconButton;
    if (button != null)
    {
        MessageBox.Show(button.Text, "Button Clicked",
            MessageBoxButton.OK);
    }
    else
    {
        var menuItem = sender as ApplicationBarMenuItem;
        MessageBox.Show(menuItem.Text, "Menu Item Clicked",
            MessageBoxButton.OK);
    }
}
```

Your click event handler checks whether the sender is a button or a menu item. Next, you retrieve the `Text` from the `sender`. A message box is shown to inform the user that the application bar item was clicked. Like any other event, the XAML editor has features that can be used to automatically create the event handler in the code behind file. This is just one of the features supported by the Windows Phone Developer tools.

10.1.2 Tooling support

Visual Studio has limited support for building an application bar with the visual and property editors. The application bar doesn't appear in the toolbox, and you can't interact with it in the visual editor. In the XAML editor, you can select the application bar, buttons, and menu items and work with their properties in the property editor (although limited). You can't use the property editor to add new buttons or menu items, and there isn't a convenient selector for picking an icon. The icons selected for the buttons aren't even shown in the visual editor.

Expression Blend has much better support for working with the application bar (see figure 10.2). The visual editor will display an application bar, complete with the appropriate icons, and will allow you to create and select the `ApplicationBar` and its buttons and menu items in the Objects and Timeline panel. The property editor for the buttons has a convenient icon selector that allows you to pick from local icons or

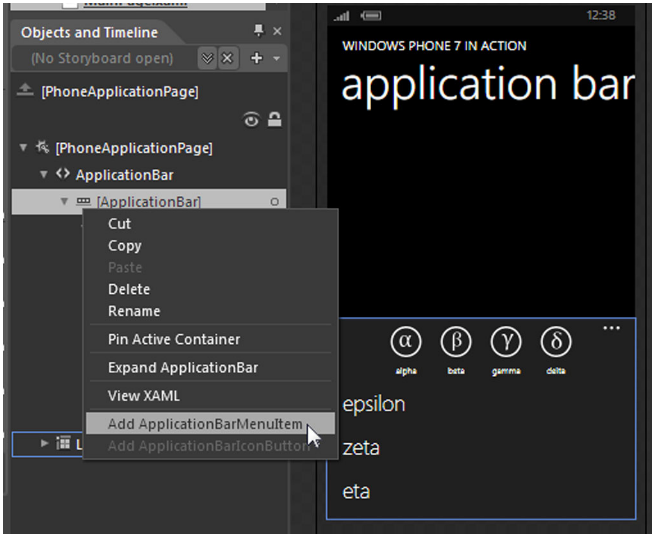

Objects and Timeline
(No Storyboard open)
[PhoneApplicationPage]
[PhoneApplicationPage]
ApplicationBar
[ApplicationBar]
Cut
Copy
Paste
Delete
Rename
Pin Active Container
Expand ApplicationBar
View XAML
Add ApplicationBarMenuItem
Add ApplicationBarIconButton
12:38
WINDOWS PHONE 7 IN ACTION
application bar
alpha
beta
gamma
delta
epsilon
zeta
eta

Figure 10.2 Expression Blend's designer support for the `ApplicationBar`

one of the icons provided in the Windows Phone SDK. When you choose an SDK icon, Blend will automatically add the icon file to the icons folder in your project.

The `ApplicationBar` and its button and menu item classes aren't `FrameworkElements` or even `DependencyObjects`. This means they don't participate in data binding, can't be located using the `FindName` API or the `VisualTreeHelper`, and don't appear in the visual tree. This also means that although you can apply an `x:Name` attribute to the buttons and menu items, and member fields will be generated for them, the fields won't automatically be wired up in `InitializeComponents`.

You can use the `x:Name` attribute to get the member variables generated and then wire them up yourself in code behind. First, add a name to the alpha button and the epsilon menu item:

```
<shell:ApplicationBarMenuItem x:Name="menuItem1"
    Text="epsilon" Click="item_Clicked" />

<shell:ApplicationBarIconButton x:Name="button1"
   IconUri="/icons/alpha.png" Text="alpha" Click="item_Clicked" />
```

Next you need to assign the generated member fields to the button and menu item instances, which you do in a new `InitializeApplicationBar` method that you add to MainPage.xaml.cs:

```
private void InitializeApplicationBar()
{
    button1 = (ApplicationBarIconButton)ApplicationBar.Buttons[0];
    menuItem1 = (ApplicationBarMenuItem)ApplicationBar.MenuItems[0];
}
```

The `ApplicationBar` instance is accessed via the `ApplicationBar` property of your `MainPage` class. The property is inherited from the `PhoneApplicationPage` class. You set your `button1` field to the first item in the `Buttons` collection. You must perform the cast to `ApplicationBarIconButton` because the `Buttons` property is an `IList`. You do the same thing for the `menuItem1` field, using the `MenuItems` collection and casting it to `ApplicationBarMenuItem`.

> **NOTE** Using a hard-coded index value to wire up the fields is fragile. If the buttons are reordered in MainPage.xaml, you could end up using the wrong button instance if the index value isn't changed in the code behind.

In this section, we've shown you how to declare and build an application bar using the Windows Phone Developer Tools. You're probably asking yourself a couple of questions. How can I change the bar and its items while an application is running? How should I alter my screen designs to accommodate the application bar?

10.1.3 Changing the application bar appearance

The application bar takes up 72 pixels of space and you need to account for that space in your page designs. You can claim more space by changing the application bar's opacity. In this situation, the application bar won't steal space from your application page, but will still be visible floating above your page. You'll need to be careful because the semi-transparent application bar might obscure your user interface. Figure 10.3 demonstrates how setting the `ApplicationBar.Opacity` property to 0.5 causes the application bar to obscure the page behind it.

Another way to reclaim screen real estate from the application bar is to use minimized mode. In minimized mode, the application bar doesn't draw any buttons, as shown in figure 10.4, and only draws the ellipses. When the user taps the control, it expands to display the icon buttons and menu. The `ApplicationBar` is placed in minimized mode by setting the `Mode` property to `ApplicationBarMode.Minimized` enumeration value. Using the `ApplicationBarMode.Default` value returns the application bar to normal. `Minimized` mode only affects how the application bar is drawn in portrait layout. The application bar is always drawn full-size in landscape layout, even when the `Mode` property is set to `Minimized`.

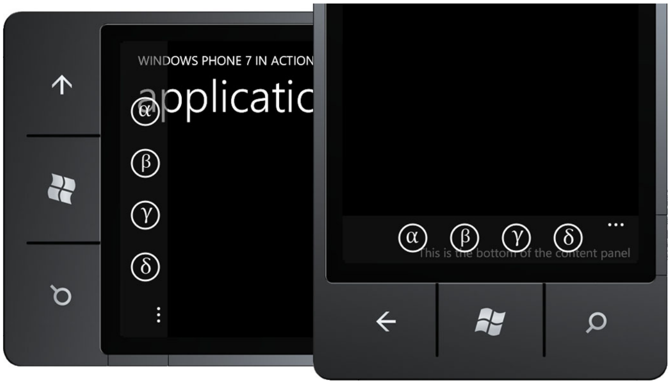

WINDOWS PHONE 7 IN ACTION
applicatio
α
β
γ
δ
This is the bottom of the content panel

Figure 10.3 Application page obscured by a partially transparent application bar

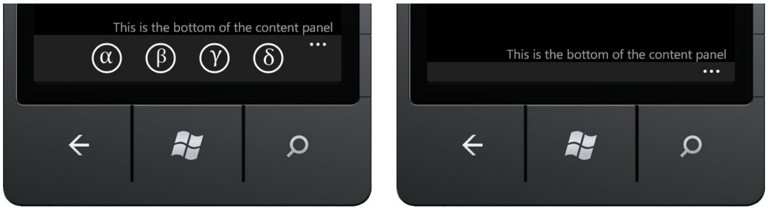

This is the bottom of the content panel
α
β
γ
δ
...
This is the bottom of the content panel
...

Figure 10.4 The application bar in default and minimized mode

The application bar contains an `IsVisible` property that can be used to show or hide the control while the application is running. This is useful if your application has some activity where the application bar isn't required, but when that activity ends, you need to display the application bar. You can demonstrate this behavior by adding a `CheckBox` to your application that can be used to control the visibility of the `ApplicationBar`:

```
<CheckBox Content="ApplicationBar.IsVisible" IsChecked="True"
    Click="appBarVisible_Clicked" />
```

In the click event handler, you set the `IsVisible` property to be the `Checkbox`'s `IsChecked` value:

```
private void appBarVisible_Clicked(object sender, RoutedEventArgs e)
{
    var checkBox = (CheckBox)sender;
    ApplicationBar.IsVisible = checkBox.IsChecked.Value;
}
```

Unlike Silverlight `UIElements`, visibility is specified using a Boolean instead of the `Visibility` enumeration.

The `ApplicationBar` also has an `IsMenuEnabled` Boolean property that controls whether the menu items are displayed when the bar is expanded. You might have some situations where you don't show the menu and other situations where you do.

10.1.4 Dynamically updating buttons and menu items

Individual buttons and menu items can be enabled or disabled at runtime using their `IsEnabled` properties. You're going to add a `CheckBox` to demonstrate changing the `IsEnabled` property of your alpha button:

```
<CheckBox Content="button1.IsEnabled" IsChecked="True"
    Click="button1Enabled_Clicked" />
```

The event handler for the check box sets the button's `IsEnabled` property to the `CheckBox`'s `IsChecked` value:

```
private void button1Enabled_Clicked(object sender, RoutedEventArgs e)
{
    var checkBox = (CheckBox)sender;
    button1.IsEnabled = checkBox.IsChecked.Value;
}
```

Though not shown here, the same technique can be used with `ApplicationBarMenuItems`. An example is available in this project's sample source code.

There might be situations where instead of disabling a button or a menu item, you'd rather remove the item from the application bar altogether. The `Buttons` and `MenuItems` collections implement `IList`'s `Add`, `Remove`, and `Insert` methods, which can be used to add and remove items from the application bar. You'll use the same `CheckBox` technique to remove or add a button from the `Buttons` collection:

```
<CheckBox IsChecked="True" Content="Show button1"
    Click="button1Show_Checked" />
```

In the event handler, you insert the button at the beginning of the collection when the check box is checked. When it's unchecked, you remove the button from the collection:

```
private void button1Show_Checked(object sender, RoutedEventArgs e)
{
    var checkBox = (CheckBox)sender;
    if (checkBox.IsChecked.Value)
        ApplicationBar.Buttons.Insert(0, button1);
    else
        ApplicationBar.Buttons.Remove(button1);
}
```

In this instance you're inserting and removing a button that was created when the page was first loaded. You could choose to destroy the button and create a brand new instance if necessary.

> **NOTE** Even though the `Buttons` collection's `Add` and `Insert` methods accept a parameter of type object, an exception will be thrown if anything other than an `ApplicationBarIconButton` instance is passed to the methods.

The user interface is updated as soon as the button is added or removed. A similar technique can be used to add and remove `ApplicationBarMenuItems`. Of course, the user won't see any changes to the menu until the application bar is expanded.

By default, the application bar is displayed in the current theme colors. If your application doesn't use the system theme, you'll likely want to change the application bar colors to match your application. The `ApplicationBar` class provides the `BackgroundColor` and `ForegroundColor` for just this situation. The application bar will automatically apply the colors to the button icons, if the icons are property designed.

10.1.5 Designing button icons

Icons should be 48 x 48 and contain only white or transparent pixels. When the dark theme is active, the application bar will display your image pretty much as is. When the light theme is active, or you're using a custom foreground color, the application bar blends all non-transparent pixels with the foreground color. You shouldn't use colored icons, as your buttons will end up with odd-looking icons. Colored icons also go against the recommendations specified in the *User Experience Design Guidelines for Windows Phone*. The application bar will automatically draw the button's bounding circle and the icons shouldn't contain the bounding circle. The icons should also fit within the bounding circle.

Text and icons for the button can be changed from code behind. You might wish to change the text and icon if you're toggling some state in your application. For example, an application that plays background music might offer a mute button and change the text and icon once the user has enabled mute.

Change your sample application to toggle `button1` between alpha and omega. First, you need to update MainPage.xaml to hook up `button1` to a new event handler:

```
<shell:ApplicationBarIconButton x:Name="button1"
    IconUri="/icons/alpha.png" Text="alpha" Click="button1_Clicked" />
```

You'll perform the text and icon change in the event handler (see the following listing). The event handler code will use the `button1` field, and will also use icon files named alpha.png and omega.png.

Listing 10.3 Toggle button icon and text

```
private void button1_Clicked(object sender, EventArgs e)
{
    if (button1.Text == "alpha")
    {
        button1.Text = "omega";
        button1.IconUri = new Uri("/icons/omega.png", UriKind.Relative);
    }
    else
    {
        button1.Text = "alpha";
        button1.IconUri = new Uri("/icons/alpha.png", UriKind.Relative);
    }
}
```

Check current state

Load icon from .xap package

The application bar will update the buttons as soon as the changes are made in code. `ApplicationBarMenuItems` can also have their `Text` property updated in code behind. The new menu item text will be visible the next time the `ApplicationBar` is expanded.

When the `ApplicationBar` is expanded or collapsed, the `StateChanged` event is raised. The event sends an `ApplicationBarStateChangedEventArgs` instance to the event handler. This event args class exposes the `IsMenuVisible` Boolean property which tells you whether the menu is visible. The `StateChanged` event is useful when you need to pause some activity in your application when the menu is shown, and resume the activity when it's hidden. You might think this event handler would be the ideal place to update the `IsEnabled` property for all your menu items, but it's not. Any changes to the menu items in the event handler won't be seen by the user until the next time the application bar is expanded.

The `ApplicationBar` and `ApplicationBarIconButtons` can be updated in the `StateChanged` event handler, and these changes will be immediately reflected in the UI. For example, you might change the foreground and background colors when the menu is opened.

The application bar is the new menu and toolbar control for Windows Phone applications. You should use the application bar in place of a row of buttons to provide access to the most common features. Less-used features should be accessed via the application bar's menu. Buttons and menu items can be declared in XAML or defined in code behind, and the application bar can dynamically update to match the state of the application.

The `ApplicationBar` is not for every application. The first page of some applications displays rich dynamic content and is the menu to the rest of the application. In this scenario, the `Panorama` control is the perfect paradigm.

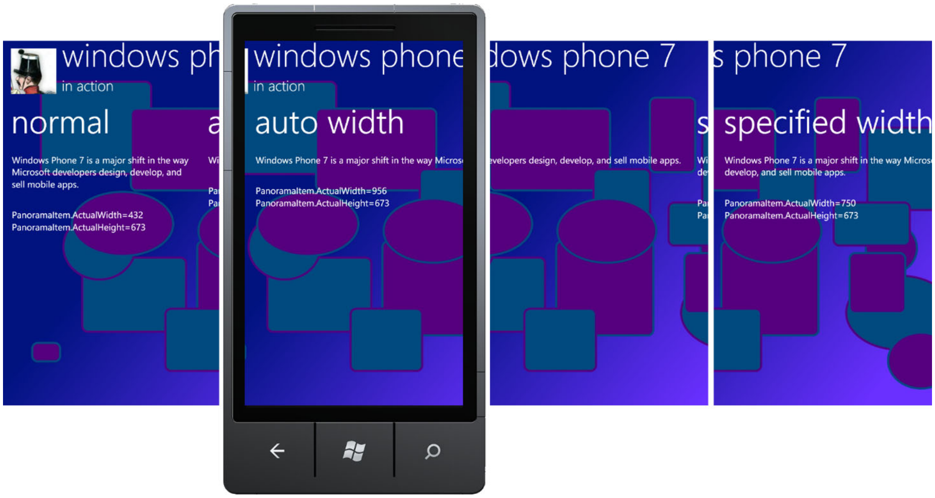
10.2 Improving the scenery with the Panorama control

One control unique to Windows Phone is the `Panorama` control, which is a long horizontal panel spread across several screens. The user pans left or right to change the viewport and move between the various screens in the control. The People Hub, the Pictures Hub, and the Music + Video Hub are just a few of the native applications that use a `Panorama` control. In this section you'll build a sample application, shown in figure 10.5, to demonstrate a few of the behaviors found only in the `Panorama` control.

A `Panorama` control should be used when you want to give a seamless flow to the contents in your application. The different viewports of the panorama are each contained within a `PanoramaItem` control.

The `Panorama` control has three different visual layers that scroll at different rates as the user pans the application. The background layer is stretched across the entire width of the control and moves at the slowest rate. The title layer moves a bit faster, but makes sure that a portion of the title appears above the current `PanoramaItem`. The top layer contains the `PanoramaItem` controls and moves at the fastest rate.

`Panorama` controls can be added to an application using either the Windows Phone Panorama Application project template, or the Windows Phone Panorama Page item template. You're going to use the Windows Phone Panorama Page item



Windows Phone 7 is a major shift in the way Microsoft developers design, develop, and sell mobile apps.

PanoramaItem.ActualWidth=432
PanoramaItem.ActualHeight=673

PanoramaItem.ActualWidth=956
PanoramaItem.ActualHeight=673

PanoramaItem.ActualWidth=750
PanoramaItem.ActualHeight=673

Figure 10.5 The sample panorama application with three items

template to embed a `Panorama` control in your application. First you need to create the sample project.

10.2.1 *Building a panorama application*

You're going to create a new project to demonstrate using a panorama control in a Silverlight application. Create a new project, named Panorama, using the Windows Phone Application template. You're not using the Windows Phone Panorama Application template, since you don't need sample Model-View-ViewModel code generated for you.

The MainPage.xaml file generated by the Windows Phone Application template isn't going to work for you, and you need to delete it from the project. You'll create a new MainPage.xaml using the Project > Add New Item menu option. From the new item dialog, choose the Windows Phone Panorama Page item template and name the new page MainPage.xaml. Once the project is created, open up MainPage.xaml and take a look at the XAML markup created by the template, which is shown in the next listing.

Listing 10.4 Panorama markup created by the item template

```
<Grid x:Name="LayoutRoot">
    <controls:Panorama Title="my application">
        <controls:PanoramaItem Header="item1">
            <Grid/>
        </controls:PanoramaItem>
        <controls:PanoramaItem Header="item2">
            <Grid/>
        </controls:PanoramaItem>
    </controls:Panorama>
</Grid>
```

❶ **Declaring panorama**

❷ **Declaring items**

MainPage.xaml was created with a `Grid` as the `LayoutRoot`, with the `Panorama` control ❶ as its only child. The `Panorama` control was generated with two placeholder `PanoramaItem` controls ❷. The content for each of the `PanoramaItems` is an empty `Grid` control. Empty grids aren't very exciting so you'll add some content shortly. First we need to discuss namespaces and assemblies.

The `Panorama` control lives in the `Microsoft.Phone.Controls` namespace, which is the same namespace that contains `PhoneApplicationPage`. The `Panorama` control doesn't live in the same assembly as `PhoneApplicationPage`, but instead can be found in `Microsoft.Phone.Controls.dll` assembly. When you used the Windows Phone Panorama Page template, a reference was automatically added to the controls assembly. Because the `Panorama` control is in a separate assembly, a separate XML namespace was added to MainPage.xaml:

```
xmlns:controls="clr-namespace:Microsoft.Phone.Controls;
➥ assembly=Microsoft.Phone.Controls"
```

Before you add some content to each of the item controls, you'll make some simple adjustments to the generated XAML. First, change the title of the `Panorama` control and give it a name:

```
<controls:Panorama x:Name="panorama" Title="windows phone 7 in action">
```

You also need to give each of the panorama items their own name and title:

```
<controls:PanoramaItem x:Name="panoItem1" Header="normal">
...
<controls:PanoramaItem x:Name="panoItem2" Header="auto width">
```

While you're at it, create a third PanoramaItem:

```
<controls:PanoramaItem x:Name="panoItem3" Header="specified width">
    <Grid/>
</controls:PanoramaItem>
```

Normally, every `PanoramaItem` would have a different set of content. In order to illustrate interesting behavior with the size of panorama items, you're going to place similar content in each `PanoramaItem`. You'll place a few text blocks in each item that will display a relatively long message, along with the width and height of the `PanoramaItem` control. The following listing details the XAML markup for the item content.

Listing 10.5 `PanoramaItem` content markup

```
<StackPanel Margin="12, 0">
   <TextBlock TextWrapping="Wrap"
            Margin="0,0,0,40">
         Windows Phone 7 is a major shift in the way Microsoft
         developers design, develop, and sell mobile apps.
   </TextBlock>
   <StackPanel Orientation="Horizontal">
      <TextBlock Text="PanoramaItem.ActualWidth=" />
      <TextBlock Text=
         "{Binding ActualWidth, ElementName=panoItem1}" />
   </StackPanel>
   <StackPanel Orientation="Horizontal">
      <TextBlock Text="PanoramaItem.ActualHeight=" />
      <TextBlock Text=
         "{Binding ActualHeight, ElementName=panoItem1}" />
   </StackPanel>
</StackPanel>
```

❶ **Relatively long message**

❷ **Bind to item's width**

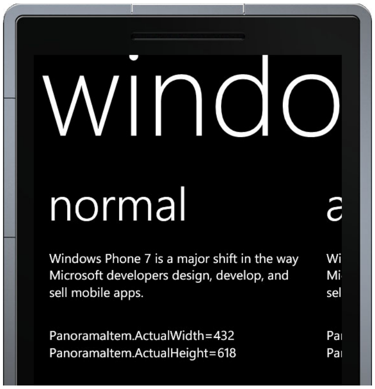
❸ **Bind to item's height**

You replace the empty `Grid` in each of the `PanoramaItem` controls with a `StackPanel` control. Inside the `StackPanel` you create a `TextBlock` to display a message ❶ that's too long to fit on a single line. You tell the `TextBlock` to wrap the text when it can't display the text on one line. Next, you create a pair of `TextBlock` controls to display the current width of the `PanoramaItem` control. You use element binding to display the `ActualWidth` property ❷ of the `PanoramaItem` control you named `panoItem1`. Finally, you display the height of the `PanoramaItem` using another pair of `TextBlock` controls, which you bind to the `ActualHeight` property ❸.

Repeat the same chunk of markup for the other two `PanoramaItem` controls. Make sure you change the element binding to `panoItem2` and `panoItem3` as appropriate. When you run the sample application now, it should appear just like the screenshot in figure 10.6.

If you drag your finger across the screen from right to left, the application will pan the screen and bring the second `PanoramaItem` into view, followed by the third `PanoramaItem`. You should notice that the title pans as well, but at a different rate from the contents. You should also notice that the message text in each panel is wrapped across three lines and that each of the panels is a single screen wide.

Remember the Pictures Hub we talked about a few pages back? The second panel in the Pictures Hub is wider than a single screen. Let's take a look at how to make your own `PanoramaItems` behave the same way.



Windows Phone 7 is a major shift in the way Microsoft developers design, develop, and sell mobile apps.

PanoramaItem.ActualWidth=432
PanoramaItem.ActualHeight=618

Figure 10.6 The Panorama sample application with fixed width items

10.2.2 Widen up the view

When the `Panorama` control lays out its children, it automatically resizes each of the `PanoramaItem` controls to fill the remaining space on the screen. After subtracting out space for the panorama title, the item header, and the overlap for the next item, a `PanoramaItem` control ends up 432 pixels wide and 618 pixels high.

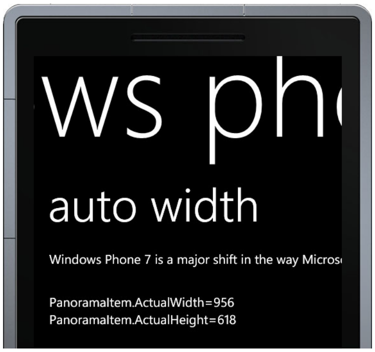
The `Panorama` control is designed so that `PanoramaItems` can have variable widths. When deciding how wide to size a `PanoramaItem` item, the `Panorama` control looks at the item's `Orientation` property. When the item's `Orientation` property is `Vertical`, the `Panorama` control will set the item's width to a single screen. The `Panorama` control allows the item to declare its own width when the `Orientation` property is set to `Horizontal`. The default value of the `Orientation` is `Vertical`, so all of your item controls are sized to a single screen.

To see this behavior in action, change `panoItem2`'s `Orientation` property to `Horizontal` and restart the application:

```
<controls:PanoramaItem x:Name="panoItem2" Header="auto width"
    Orientation="Horizonatal">
```

Pan over to the auto width panel, and you should notice that the message text is now in a single line, and the panel spans across a couple of screens. Figure 10.7 shows the updated panel.

The `TextBlock` containing the message text prefers to display the message in a single line. When the `Panorama` layout routine asks the `PanoramaItem` for its preferred width, the `TextBlock`'s preferred size is reported back. The result is that the `PanoramaItem` has an `ActualWidth` of 956 pixels. If the message text were longer, the panel would be wider; the longer the text, the wider the panel.



Windows Phone 7 is a major shift in the way Micros

PanoramaItem.ActualWidth=956
PanoramaItem.ActualHeight=618

Figure 10.7 A wide PanoramaItem that calculates its own width

Infinitely wide panels are undesirable. Just like any other layout scenario, you can control the width of a panel by setting either the `Width` or `MaxWidth` properties of the `PanoramaItem` control. Hard-code the width of `panoItem3` to 750 pixels:

```
<controls:PanoramaItem x:Name="panoItem3" Header="specified width"
    Orientation="Horizonatal" Width="750" >
```

If you run the application now, you should see two lines of message text on the third panel. Once you're on the third panel, use the Start button to exit the application and return to it using the Back button. Now exit the application, change the project property so that the application is tombstoned upon deactivation, and restart the application. Scroll to the third panel, press the Start button and once again return to it using the Back button. The application restores, but the selected panel is the first one instead of the third. Let's look at how you can restore the panel that was selected before the application was tombstoned.

10.2.3 Remembering where you are

Well-behaved applications remember their state when the user switches to another application, and they restore the state when the application is reactivated. Panorama applications are no different, and should return the user to the correct `PanoramaItem` when the application is restarted.

The `Panorama` control exposes `SelectedIndex` and `SelectedItem` properties, but they're both read-only, and can't be used to restore state. Instead, the `Panorama` control provides the `DefaultItem` property. Before you can use the `DefaultItem` property to restore user state on reactivation, you need to record which `PanoramaItem` control is selected.

When the user pans to a new item, the `Panorama` control fires the `SelectionChanged` event. Wire up the event in MainPage.xaml:

```
<controls:Panorama x:Name="panorama" Title="windows phone 7 in action"
    SelectionChanged="panorama_SelectionChanged">
```

Implement the event handler in MainPage.xaml.cs by saving the selected index to application settings:

```
private void panorama_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    IsolatedStorageSettings.
        ApplicationSettings["selection"] = panorama.SelectedIndex;
}
```

You restore the selection in the `OnNavigatedTo` event handler where you read the selected index from application settings, and use it to set the `DefaultItem` property:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    int selectedIndex;
    if (IsolatedStorageSettings.ApplicationSettings
        .TryGetValue("selection", out selectedIndex))
    {
        panorama.DefaultItem = panorama.Items[selectedIndex];
    }
}
```

When the `DefaultItem` property is set in code, the `Panorama` immediately makes the specified `PanoramaItem` control the selected item. The change in selection isn't animated like it is when the user pans the screen. This could be disconcerting for the user if you have logic in your application that changes the `DefaultItem` based on non-panning activity, such as a button click.

There's one other idiosyncrasy with using the `DefaultItem` property. The `Panorama` title is lined up with the item that's specified as the default. Figure 10.8 illustrates how the title is lined up with the second `PanoramaItem` control when it's the default item.

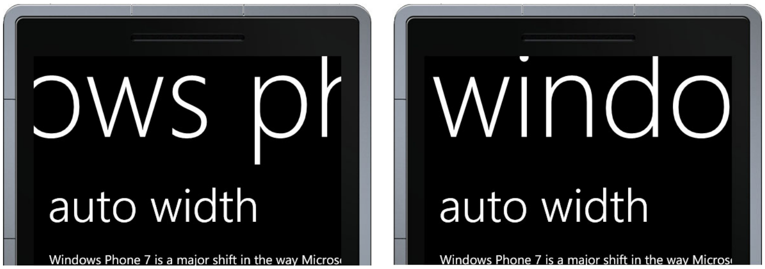
When the normal panel is the default item, the auto width panel aligns with the letter O in the word Windows. When the auto width panel is the default, it aligns with the letter W in Windows. The default panel also is aligned to the left edge of the background image, when a background is specified. You'll add a background image to your Panorama control next.

10.2.4 Adding a background

When we first described the `Panorama` control, we mentioned that there were three panning layers that moved at different speeds. You've only seen two of these layers in action so far as you haven't yet added a background to your application. The `Panorama` Background property is inherited from the `Control` class, and is specified just like the background of any other control.

The `Background` property is of type `Brush`. This means the background can be painted with a solid color or one of the gradient brushes. You can also use an `ImageBrush` or even a `VideoBrush`.

NOTE `VideoBrush` is only available with the Windows Phone SDK 7.1.



Windows Phone 7 is a major shift in the way Micros

Windows Phone 7 is a major shift in the way Micros

Figure 10.8 Title alignment when the second item isn't the default (left) and when it is the default (right)

Creating the `ImageBrush` in XAML is easy:

```
<controls:Panorama x:Name="panorama" Title="windows phone 7 in action"
    SelectionChanged="panorama_SelectionChanged">
    <controls:Panorama.Background>
        <ImageBrush ImageSource="PanoramaBackground.jpg" />
    </controls:Panorama.Background>
```

In this snippet, you're telling the `ImageBrush` to use the file named PanoramaBackground.jpg. You should add the file to the root of your project and select a build action of `Resource`, which causes the image to be compiled into your assembly. This is the optimal option for `Panorama` backgrounds because the image is available as soon as the control is displayed. Background images can be loaded from the .xap file content or even from an internet location, but the user will experience a delay between when the `Panorama` control is displayed and when the background image first appears.

NOTE For the best performance and user experience, choose a background image that is between 480 x 800 pixels and 1024 x 800 pixels.

All user interface elements pick up their look and feel from the system theme resources. This means that if the phone is set to the dark theme, the background will be black and text displayed in the `Panorama` will be white. The text will be black and the background white when the user has chosen the light theme for their phone.

Unless you explicitly set the text color on the `Panorama` control, you must ensure that the default system text color is readable against your background image. The image in this PanoramaBackground.jpg example doesn't play well with black text. To remedy the situation, you hard-code the `Panorama` `Foreground` property to `White`:

```
<controls:Panorama x:Name="panorama" Title="windows phone 7 in action"
    SelectionChanged="panorama_SelectionChanged"
    Foreground="White">
```

By setting the `Foreground` property, you're telling the `Panorama` control to ignore the system text color.

Now that you have a background image, you're nearly done with your sample application. You have `PanoramaItems` with different widths and you restore the active panel when the application is re-launched. Before we move on to the `Pivot` control, let's take a deeper look at panorama titles.

10.2.5 Customize the title

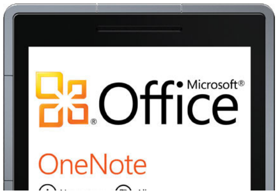


Figure 10.9 The Office Hub with an image in the panorama title

When you look through the various built-in hubs on the phone, you'll notice that most use simple text for the panorama title. One notable exception is the Office Hub. Shown in figure 10.9, the Office Hub displays an image in the panorama title.

The `Title` property of the `Panorama` control is of type `Object`, and it's displayed with the data template specified in the `TitleTemplate` property. You aren't going to replace the template in this example because you're just going to add a `Grid` control with an `Image` and two `TextBlocks`. The XAML markup for the new panorama title is shown in the next listing.

Listing 10.6 Adding an image and two title lines to the `Panorama` control

```
<controls:Panorama x:Name="panorama"
        Foreground="White"
        SelectionChanged="panorama_SelectionChanged">
    <controls:Panorama.Title>
        <Grid Margin="0,55,0,0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>
            <Image Source="PanoramaLogo.png"
                Grid.RowSpan="2" Margin="12,36,12,0" />
            <TextBlock Grid.Column="1" Text="windows phone 7"
               FontSize="{StaticResource PhoneFontSizeExtraExtraLarge}" />
            <TextBlock Grid.Row="1" Grid.Column="1" Text="in action"
                FontSize="{StaticResource PhoneFontSizeLarge}" />
        </Grid>
    </controls:Panorama.Title>
```

❶ **Use property element syntax**

❷ **Divide into 2 rows and 2 columns**

❸ **Add image**

❹ **Add title lines**

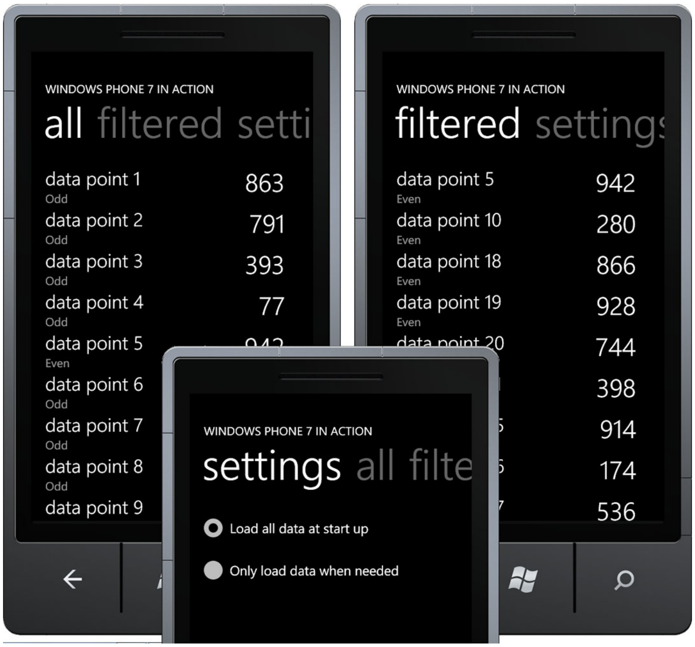
Using XAML's property element syntax ❶, you add a `Grid` control to the `Panorama.Title` property. The `Grid` is divided into two rows and two columns ❷. Next you declare the `Image` control, placing it in the first two rows of the first column of the `Grid` ❸. The `Image` control will display the image in the file PanoramaLogo.png, which you add to the project with a build action of `Resource`. Finally, you add two `TextBlock` controls that display the title ❹. You bind the `FontSize` property of each control to a theme font size resource.

Run the application and note the scrolling behavior of the new title. The image and two text lines scroll as a single unit when the user pans across the panorama. The continuous multi-layer movement of the background, title, and content is unique to the `Panorama` control. Multi-layer movement distinguishes the `Panorama` control from its counterpart, the `Pivot` control.

10.3 Pivoting around an application

The `Pivot` control is the Windows Phone equivalent to a tab control. The `Pivot` control displays the title of each of its child `PivotItems` across the top of the control. The user can switch between items by tapping the titles or panning to scroll a new pivot into view. The main title displayed by the `Pivot` control is stationary and doesn't move.

Unlike the `Panorama` control, which loads every item when it's created, the `Pivot` control only loads the currently displayed page. When a user switches to another pivot



WINDOWS PHONE 7 IN ACTION
all filtered setti
data point 1 863
Odd
data point 2 791
Odd
data point 3 393
Odd
data point 4 77
Odd
data point 5
Even
data point 6
Odd
data point 7
Odd
data point 8
Odd
data point 9

WINDOWS PHONE 7 IN ACTION
filtered settings
data point 5 942
Even
data point 10 280
Even
data point 18 866
Even
data point 19 928
Even
data point 20 744
398
914
174
536

WINDOWS PHONE 7 IN ACTION
settings all filte
◉ Load all data at start up
○ Only load data when needed

Figure 10.10 The sample Pivot application with three pivots

item, the old item is unloaded and removed from the visual tree. We'll examine the events raised when `Pivot` switches between items.

To demonstrate the features of the `Pivot` control you'll build a new sample application. Shown in figure 10.10, this application will contain three `PivotItems` implementing a pattern that's common in many pivot-based experiences. The first pivot item displays an unfiltered list of data. The second pivot item displays the same list, but filtered to a subset of the data. The last pivot item allows the user to specify options or settings for the application.

`Pivot` controls can be added to an application using either the Windows Phone Pivot Application project template, or the Windows Phone Pivot Page item template. You're going to use the Windows Phone Pivot Page item template to embed a `Pivot` control in your application. First you need to create the sample project.

10.3.1 Building the sample

You're going to create another new project to demonstrate using a `Pivot` control as the main page of a Silverlight application. Create a new project, named Pivot, using the Windows Phone Application template. You're starting with the basic application instead of the Windows Phone Pivot Application template. You're not using the pivot application template since you don't need sample Model-View-ViewModel code generated for you.

The MainPage.xaml file generated by the Windows Phone Application template isn't going to work here, so you need to delete it from the project. You'll create a new

MainPage.xaml using the Project > Add New Item menu option. From the new item dialog, choose the Windows Phone Pivot Page item template and name the new page MainPage.xaml. Once the project is created, open up MainPage.xaml and take a look at the XAML markup created by the template, which is shown in the following listing.

Listing 10.7 Pivot markup created by the item template

```
<Grid x:Name="LayoutRoot" Background="Transparent">
    <controls:Pivot Title="MY APPLICATION">
        <controls:PivotItem Header="item1">
            <Grid/>
        </controls:PivotItem>
        <controls:PivotItem Header="item2">
            <Grid/>
        </controls:PivotItem>
    </controls:Pivot>
</Grid>
```

❶ Declaring pivot

❷ Declaring items

MainPage.xaml was created with a `Grid` as the `LayoutRoot`, with a `Pivot` control ❶ as its only child. The `Pivot` control was generated with two placeholder `PivotItem` controls ❷. The content for each of the `PivotItems` is an empty `Grid` control.

Before you add some content to each of the item controls, you'll make some simple adjustments to the generated XAML. First you change the title of the `Pivot` control and give it a name:

```
<controls:Pivot x:Name="pivot" Title="WINDOWS PHONE 7 IN ACTION">
```

You also need to give each of the pivot items their own name and title:

```
<controls:PivotItem x:Name="allDataItem" Header="all">
...
<controls:PivotItem x:Name="filteredItem" Header="filtered">
```

While you're at it, create a `PivotItem` for the settings pivot:

```
<controls:PivotItem Header="settings">
    <StackPanel>
        <RadioButton x:Name="allDataOption" IsChecked="True"
            Content="Load all data at start up" />
        <RadioButton x:Name="asNeededOption"
            Content="Only load data when needed" />
    </StackPanel>
</controls:PivotItem>
```

Providing the user an option on the settings pivot implies that you need to save and reload the selected option when the application restarts. You'll save the option in the `OnNavigatedFrom` method override:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["loadAllData"]
        = allDataOption.IsChecked.Value;
}
```

You reload the save option in the `OnNavigatedTo` method override:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    bool loadAllData = false;
    IsolatedStorageSettings.ApplicationSettings
        .TryGetValue("loadAllData", out loadAllData);
    allDataOption.IsChecked = loadAllData;
    asNeededOption.IsChecked = !loadAllData;
}
```

Your sample application is off to a good start. You have three pivots and your settings page remembers the options chosen by the user. You should also remember the currently selected `PivotItem` when the user switches away from and then back to your application.

10.3.2 Remembering the current selection

You should be a good citizen and restore the selected pivot when the user switches tasks, and be sure to return to the appropriate pivot when the application is reactivated. Add the following line to the `OnNavigatedFrom` method override:

```
State["selection"] = pivot.SelectedIndex;
```

You want your application to always start a new instance showing the first pivot item, so you're going to store the selection in the `State` dictionary instead of application settings.

Restoring the selection isn't as straightforward as saving it. In some situations, the `Pivot` control won't allow the `SelectedIndex` to be modified before it's loaded. To prevent problems, you'll add your index restoration logic to an event handler wired up to the `Pivot` control's `Loaded` event:

```
<controls:Pivot x:Name="pivot" Title="WINDOWS PHONE 7 IN ACTION"
    Loaded="pivot_Loaded">
```

In the implementation of the event handler you check whether the `State` dictionary contains a selection, and if so, use it to change the current pivot:

```
private void pivot_Loaded(object sender, RoutedEventArgs e)
{
    if(State.ContainsKey("selection"))
    {
        pivot.SelectedIndex = (int)State["selection"];
    }
}
```

When the `SelectedIndex` is changed from code, the user will see the normal animation as the `Pivot` control moves the related `PivotItem` to the foreground.

The user can now safely switch applications knowing that their pivot selection will be restored when they switch back to the application. At this point, there's no good reason to switch back since the application doesn't display any data.

10.3.3 Generating sample data

Your sample application is modeled after a class of data-browsing applications that use a `Pivot` control to move between different sets of filtered data. Without data, you're going to have a hard time demonstrating a filtering technique. For simplicity, your application is going to generate a sample dataset. In a real application, the data set might come from a file, a database, or a web service. Add a new class file to the project and name it SampleData.cs. The next listing shows the implementation of the `SampleData` class.

Listing 10.8 Generating sample data with the `SampleData` class

```
public enum SampleCategory{ Even, Odd }

public class SampleData
{
    public string Name { get; set; }
    public int Value { get; set; }
    public SampleCategory Category { get; set; }

    public static IEnumerable<SampleData> GenerateSampleData()
    {
        var results = new List<SampleData>();
        var generator = new Random();

        for (int i = 1; i < 100; i++)
        {
            var value = generator.Next(1000);
            var data = new SampleData
            {
                Name = "data point " + i,
                Value = value,
                Category = value % 2 == 0 ?
                    SampleCategory.Even : SampleCategory.Odd,
            };
            results.Add(data);
        }
        return results;
    }
}
```

❶ **Enum to distinguish odd from even**

❷ **Sample properties**

❸ **Generate 100 random data points**

Within the SampleData.cs file you create a class named `SampleData` and an enum named `SampleCategory` ❶. The category enumeration value will be used later when you build the filter. The `SampleData` class has three properties for `Name`, `Value`, and `Category` ❷. You also created a static method that you can call to generate 100 random values ❸. As each data point is created, you check whether the random value is even or odd and assign the appropriate category.

Next, update MainPage.xaml.cs by adding a field to hold the generated data, and initialize the field in the class constructor:

```
IEnumerable<SampleData> data;

public MainPage()
{
    InitializeComponent();
    data = SampleData.GenerateSampleData();
}
```

Now that you have some data to display, you need something that will display data. A `ListBox` should do nicely, and you'll add one to each `PivotItem`. Both `ListBoxes` will display data exactly the same way and you can create a single `DataTemplate` that can be shared. The `DataTemplate`, shown in the following listing, will be added to the page's `Resources` dictionary.

Listing 10.9 The `DataTemplate` used to display `SampleData`

```
<phone:PhoneApplicationPage.Resources>
    <DataTemplate x:Key="dataTemplate">
        <Grid Width="432">
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <TextBlock Text="{Binding Name}"
                Style="{StaticResource PhoneTextLargeStyle}" />
            <TextBlock Grid.Row="1" Text="{Binding Category}"
                Style="{StaticResource PhoneTextSubtleStyle}" />
            <TextBlock Grid.RowSpan="2" Grid.Column="1"
                Text="{Binding Value}" HorizontalAlignment="Right"
                Style="{StaticResource PhoneTextExtraLargeStyle}" />
        </Grid>
    </DataTemplate>
</phone:PhoneApplicationPage.Resources>
```

❶ **Create 2 x 2 grid**

❷ **Value spans 2 rows**

The `DataTemplate` displays the `SampleData` using three `TextBlock` controls placed in a `Grid`. The `Grid` is divided into two rows and two columns ❶. Each of the three `TextBlocks` uses a different style defined in the system theme resources. The `TextBlock` displaying the value spans both rows in the second column ❷.

Now add the `ListBoxes`. In both the all and filtered `PivotItems`, replace the empty `Grid` control with a `ListBox`. Bind the `ListBox`'s `ItemTemplate` to the `DataTemplate` you just added:

```
<controls:PivotItem x:Name="allDataItem" Header="all">
    <ListBox x:Name="allDataList"
        ItemTemplate="{StaticResource dataTemplate}" />
</controls:PivotItem>
```

We only show the XAML for the first pivot, but you need to add the same markup to the filtered `PivotItem`, giving the `ListBox` the name `filteredDataList`.

Next you need to load data into the `ListBoxes`. The `pivot_Loaded` method is a good place to do this. Add the following code to the bottom of the `pivot_Loaded` method:

```
if (allDataOption.IsChecked.Value)
{
    allDataList.ItemsSource = data;
    filteredDataList.ItemsSource = from d in data
                                   where d.Category == SampleCategory.Even
                                   select d;
}
```

If the user has selected to load all data at startup, you set the `ItemsSource` of the first `ListBox` to the data collection. You filter the data using a LINQ expression and set the result as the `ItemsSource` of the second `ListBox`. When you run the application now with the load at startup option, you'll see a list of 100 items in the first `PivotItem`, and a list of even-valued data points in the second `PivotItem`.

There are a couple of things you should notice. When the application first starts, the data in the first `PivotItem` appears after a slight delay. When you move between the pivots, the data appears instantaneously. The delay is due to waiting to set the `ItemsSource` property until after the `Pivot` is loaded. After the initial load, the data appears instantaneously because both `ListBoxes` are holding the data in memory as well as the UI elements needed to display the data. The data is held in memory, even if the user never visits the `PivotItem`.

Holding a large amount of data and user interface elements in memory could create performance and resource problems for an application. The `Pivot` control provides developers a set of events so that they manage application resources and dynamically load and unload pages.

10.3.4 Dynamically loading pages

To enable developers to control when data is loaded and discarded, the `Pivot` control provides a series of events. Two events, named `LoadingPivotItem` and `LoadedPivotItem`, are raised when a `PivotItem` is gaining focus. Two complementary events, `UnloadingPivotItem` and `UnloadedPivotItem`, are raised when a `PivotItem` is losing focus. A fifth event, `SelectionChanged`, is used to determine which item is being selected, and which item is losing selection.

The events are raised in the order showing in figure 10.11. You should notice that the `SelectionChanged` event is raised after the new item begins loading but before the old item starts unloading.

The `LoadedPivotItem` and `UnloadedPivotItem` events are the perfect place for you to load and unload your data. Wire the `Pivot` control events to new event handlers in your code behind:

```
<controls:Pivot x:Name="pivot" Title="WINDOWS PHONE 7 IN ACTION"
    Loaded="pivot_Loaded" LoadedPivotItem="pivot_LoadedPivotItem"
    UnloadedPivotItem="pivot_UnloadedPivotItem">
```

The next listing details the `LoadedPivotItem` event handler and how you load data into the `ListBox`.

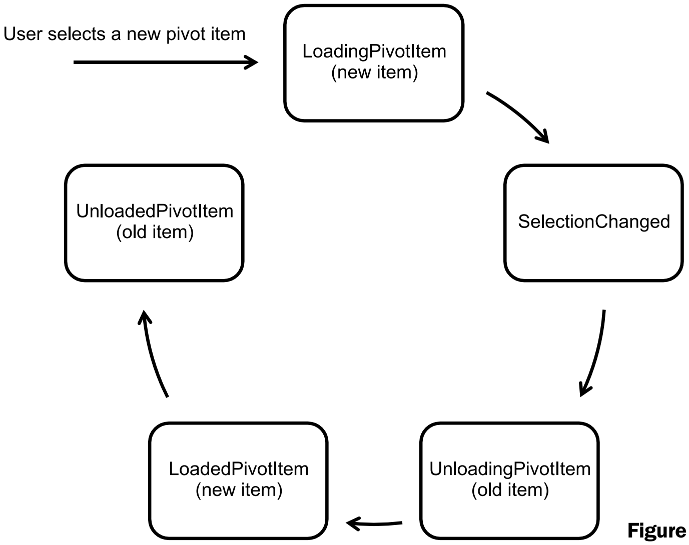

User selects a new pivot item
LoadingPivotItem (new item)
SelectionChanged
UnloadingPivotItem (old item)
LoadedPivotItem (new item)
UnloadedPivotItem (old item)

Figure 10.11 Pivot selection events

Listing 10.10 Loading `ListBox` data in response to a `LoadedPivotItem` event

```
private void pivot_LoadedPivotItem(object sender, PivotItemEventArgs e)
{
    if (e.Item == allDataItem && allDataList.ItemsSource == null)
    {
        allDataList.ItemsSource = data;
    }
    else if (e.Item == filteredDataItem
        && filteredDataList.ItemsSource == null)
    {
        filteredDataList.ItemsSource =
            from d in data
            where d.Category == SampleCategory.Even
            select d;
    }
}
```

Is data already loaded? ❶

The `LoadedPivotItem` event passes a `PivotItemEventArgs` class, which exposes a single `Item` property. The `Item` property is a reference to the `PivotItem` that's currently being selected. You check whether the `Item` being loaded is the `allDataItem` and confirm that the `allDataList` doesn't currently contain data ❶. If all is well, you set the `ItemsSource` property just like you did when loading data during startup. You perform the equivalent check and set whether the current `Item` is the filtered `PivotItem`.

Unloading data when the user moves to another `PivotItem` follows a similar pattern. The `UnloadedPivotItem` event is handled by the `pivot_UnloadedPivotItem` method, shown in the following listing.

Listing 10.11 Clearing the `ItemsSource` property when an item is unloaded

```
private void pivot_UnloadedPivotItem(object sender, PivotItemEventArgs e)
{
    if (!allDataOption.IsChecked.Value)
    {
        if (e.Item == allDataItem)
        {
            allDataList.ItemsSource = null;
        }
        else if (e.Item == filteredDataItem)
        {
            filteredDataList.ItemsSource = null;
        }
    }
}
```

① **Load data as needed?**

② **Clear ListBox**

First you check whether the user has selected the option to load data only as needed ①. When in the only-as-needed mode, you clear the `ListBox` when the `PivotItem` containing it is unloaded. You clear the `ListBox` by setting its `ItemsSource` property to `null`.

Your sample application is now complete. Data is loaded and shown in two views, one of them filtering out all the odd data. The user can specify how you handle the data by listening for loaded and unloaded events raised by the `Pivot` control. You use the unloaded event handlers to clean up resources used by the list boxes in the pivot items.

10.4 Summary

In this chapter you learned about controls that are new to Windows Phone and aren't available in Silverlight for the browser applications. The `ApplicationBar`, `Panorama`, and `Pivot` controls are only found in the Windows Phone SDK.

You learned how to use the application bar as a toolbar and menu for your application. We showed you how the `Panorama` control creates a unique user experience by employing three layers of movement when scrolling between items in the control. You built an application that uses a `Pivot` control to implement a data filtering pattern common to many applications.

Applications can mix and match the controls presented in this chapter to build compelling applications. Many applications employ a `Panorama` control for the main page, but use `Pivot` controls once the user drills into the content of the application. Panorama and pivot items can serve as the host container for a list box. `Pivot` controls often provide an application bar with buttons that let the user manage data displayed in the control.

In the next chapter you learn how common Silverlight controls have been modified to work on the phone. You also learn techniques for ensuring that controls respect the light and dark system themes, and use the accent color specified by user preferences. We'll also introduce you to the Silverlight Toolkit for Windows Phone, which provides a few additional controls unique to the phone that are missing from the Windows Phone SDK.

11

Building Windows Phone UI with Silverlight controls

This chapter covers

- Using theme-aware controls
- Specifying keyboard layouts
- Using render transforms
- Working with the Silverlight Toolkit

Silverlight for Windows Phone 7 provides several user interface controls to application developers. Though most of the controls exist in Silverlight for the browser and other user interface libraries, this chapter addresses features of the controls that are relevant to building user interfaces for the Windows Phone. We discuss which controls to use to prompt the user for input, which controls to use to display output to the user, and techniques to combine or to modify controls in order to create new user experiences. You'll also learn how to ensure the controls you use integrate with the user-specified system theme.

In addition to the controls shipped as part of the Windows Phone SDK, we'll look at a few of the controls that are available in the Silverlight Toolkit for Windows Phone. The Silverlight Toolkit is an open source project hosted on CodePlex. Microsoft uses the Silverlight Toolkit project to share new controls and components with the development community.

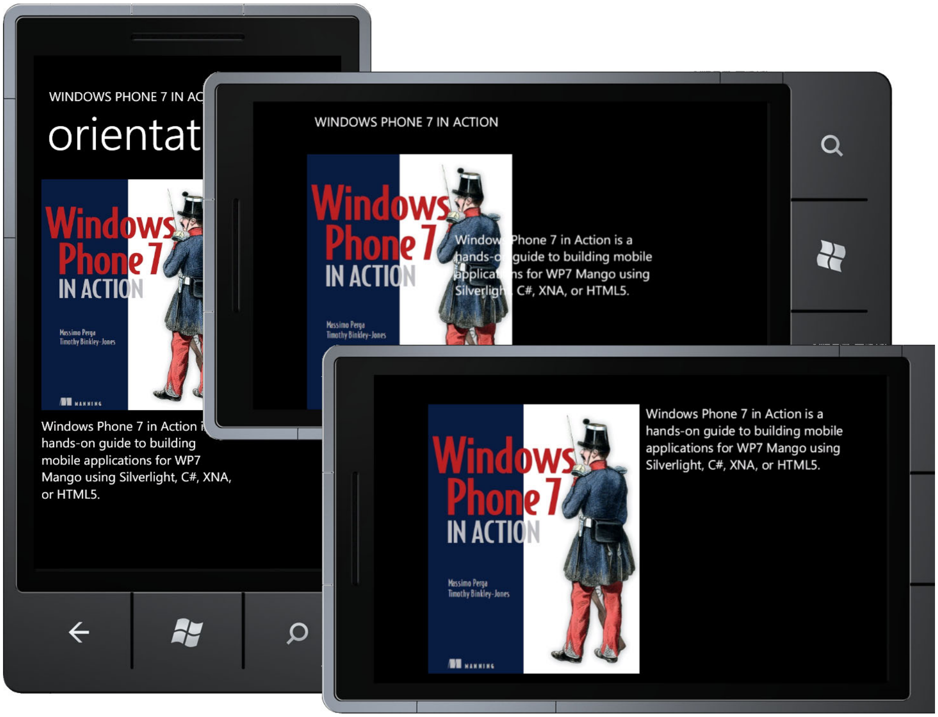
Before we get into the nuts and bolts of the controls, we show you how to build an application that uses different screen layouts when the user turns the phone from portrait to landscape orientation. The orientation sample application uses Silverlight's `VisualStateManager` to animate transitions from portrait to landscape.

11.1 Handling page orientation

Windows Phones, as with other small form factor devices, are able to deal both with landscape and portrait mode. Each page in the application must declare the orientation modes it supports, so that the runtime framework can properly display the application and notify the application when the orientation changes.

Let's consider a situation where the landscape and portrait view of a page aren't identical. You can see an example of this situation by running the calculator application that's shipped with every Windows Phone. In portrait orientation, the calculator presents basic arithmetic operations. When you switch the phone to landscape orientation, the calculator adds buttons for trigonometric, logarithmic, and other mathematical operations to the user interface.

In this section you're going to build a sample application, shown in figure 11.1, which displays information about this book. In portrait orientation, the cover image is shown above the book's description. In landscape orientation, the application and



Windows Phone 7 in Action is a hands-on guide to building mobile applications for WP7 Mango using Silverlight, C#, XNA, or HTML5.

Figure 11.1 The OrientationWithStates sample application demonstrating the transition between portrait and landscape orientation

page titles are hidden, and the book description is placed to the right of the cover image. As a bonus, the changes to the layout are animated so the user can see the description moving from one position to the other.

To start the new sample application, create a new project from the Windows Phone Application project template and name the project OrientationWithStates. By default, the main page of a project generated by the Windows Phone Application template only supports portrait orientation. Let's take a closer look at how a page declares support for portrait and landscape orientations.

11.1.1 Supported orientations

The way in which a Silverlight page declares its supported orientations is by setting the `PhoneApplicationPage`, `SupportedOrientations`, and `Orientation` properties, either in XAML or code-behind:

```
<phone:PhoneApplicationPage ...
    SupportedOrientations="Portrait" Orientation="Portrait" >
```

In this case, the page declares that it supports only `Portrait` mode using the `SupportedOrientations` attribute. `SupportedOrientations` can be a set of one of the following values:

- `Portrait`
- `PortraitOrLandscape`
- `Landscape`

The `Orientation` attribute defines the default orientation of the page. If just one orientation is supported, then the `Orientation` property must match the `SupportedOrientations` property; otherwise it can be one of the `PageOrientation` values:

- `None`
- `Portrait`
- `Landscape`
- `PortraitUp`
- `PortraitDown`
- `LandscapeLeft`
- `LandscapeRight`

The `PageOrientation` enum values are bit fields. The `PortraitUp` value has its bit flag for `Portrait` set. `LandscapeLeft` and `LandscapeRight` also have the bit flag for `Landscape` set. `PortraitDown` isn't used since the Windows Phone won't rotate the screen into an upside-down position.

If you've assigned a page's `SupportedOrientations` as `PortraitOrLandscape`, you have to handle both orientations. For many applications, the best way to support both orientations is by using a layout control like `StackPanel` or `Grid`, and doing nothing else. When the phone changes orientation, the layout panels automatically resize themselves and adjust the positions of their child controls.

To see how a layout panel like `Grid` automatically readjusts its children, you only need to change the sample application and modify MainPage.xaml to support both `Landscape` and the `Portrait` mode:

```
SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
```

When you launch the modified application, the layout can be changed by pressing one of the rotation buttons on the emulator's command bar. When the emulator changes orientation, the page automatically updates. This is because the page uses a combination of `Grid` and `StackPanel` controls, which rearrange their child controls. This works well when both orientations show the same basic user interface.

11.1.2 Animating orientation transitions

The OrientationWithStates sample application displays the book's cover in an `Image` control, and the book's description in a `TextBlock`. To allow finer control over the placement of the controls, you need to change the `ContentPanel` control to be a `Canvas` control, as seen in the following snippet. You also need to import a picture into the root of the project, and set its build action property to `Content`. The final markup for the `ContentPanel` in MainPage.xaml is shown in the following snippet:

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0" >
  <Image x:Name="coverImage" Source="/cover.jpg" Canvas.Left="0"
    Canvas.Top="0" Width="300" Height="360" Stretch="Fill" />
  <TextBlock x:Name="coverText" Canvas.Left="0" Canvas.Top="370"
    Width="300" Height="360" TextWrapping="Wrap" Text="Windows Phone 7
➥  in Action is a hands-on guide to building mobile applications for
➥  WP7 Mango using Silverlight, C#, XNA, or HTML5."  />
</Canvas>
```

You're going to use visual states and the `VisualStateManager` to transition between portrait layout and landscape layout in the sample application. The `VisualStateManager` allows you to define the look and feel of your user interface for a set of circumstances or states. In the case of this sample application there are two states—`Portrait` and `Landscape`. If you're not familiar with visual states and the `VisualStateManager`, you can read more about them in Pete Brown's book *Silverlight 5 in Action*.

You've already designed your `ContentPanel` just as you want it to appear in portrait orientation, and the corresponding visual state doesn't need to be adjusted. When the phone is switched to landscape orientation, you need to perform three adjustments—hide the title panel, resize the cover image, and resize and move the description text. The following listing contains the visual state markup necessary to perform these steps.

Listing 11.1 Defining visual states for portrait and landscape orientation

```
<Grid x:Name="LayoutRoot" Background="Transparent">
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="OrientationStates">

      <VisualState x:Name="Portrait" />
```

① **Portrait is default state**

```
        <VisualState x:Name="Landscape">                    Hide title panel ❷
          <Storyboard>
            <DoubleAnimation To="0" Storyboard.TargetName="TitlePanel"
              Storyboard.TargetProperty="Height" Duration="00:00:00.25" />

            <DoubleAnimation To="420" Duration="00:00:00.25"
              Storyboard.TargetName="coverImage"
              Storyboard.TargetProperty="Height" />              ❸ Resize
            <DoubleAnimation To="330" Duration="00:00:00.25"       cover
              Storyboard.TargetName="coverImage"                   image
              Storyboard.TargetProperty="Width" />

            <DoubleAnimation To="420" Duration="00:00:00.25"
              Storyboard.TargetName="coverText"
              Storyboard.TargetProperty="Height" />
            <DoubleAnimation To="330" Duration="00:00:00.25"
              Storyboard.TargetName="coverText"
              Storyboard.TargetProperty="Width" />
            <DoubleAnimation To="0" Duration="00:00:00.25"
              Storyboard.TargetName="coverText"
              Storyboard.TargetProperty="(Canvas.Top)" />        ❹ Move
            <DoubleAnimation To="340" Duration="00:00:00.25"       description
              Storyboard.TargetName="coverText"
              Storyboard.TargetProperty="(Canvas.Left)" />
          </Storyboard>
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
...
</Grid>
```

You start by adding a `VisualStateMananger.VisualStateGroups` element to the `LayoutRoot Grid` control with one `VisualStateGroup` named `OrientationStates`. The first `VisualState` in the group, named `Portrait` ❶, is the default group and doesn't contain any animations. The second `VisualState`, named `Landscape`, contains several animations that are played when the landscape visual state is activated. The first animation shrinks the `Height` of the `TitlePanel` to zero ❷. The next two animations ❸ change the `Height` and `Width` properties of the `coverImage Image` control. The `Height` and `Width` of the `coverText TextBlock` are adjusted with two more animations. Finally, the `Top` and `Left` properties of the `coverText TextBlock` are changed so that the `TextBlock` appears to the right of the `Image` control instead of below it ❹.

To enable the title panel transition animation, you need to make one minor adjustment to the title panel's markup. You need to explicitly set the `Height` property:

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28"
    Height="116">
```

When the `Landscape` visual state is activated, the animations run and the screen is updated. You've set the animation duration to be one quarter of a second. When the `Portrait` visual state is activated, the animations are run in reverse. The visual states are activated with a call to the `GoToState` method of the `VisualStateManager`. In the next section we show you how to call the `GoToState` method with the `Orientation` property changes.

11.1.3 Changing orientation

When the phone is turned from a portrait orientation to a landscape orientation, the movement is detected by the application framework, and the current page's `Orientation` property is updated. A running application can detect changes in the `Orientation` property by subscribing to the `OrientationChanged` event provided by the `PhoneApplicationPage` class.

In this sample application, you want to wire up the `OrientationChanged` event to an event handler in the `MainPage` class. Add the following markup to the `PhoneApplicationPage` element in MainPage.xaml:

```
<phone:PhoneApplicationPage ...
    OrientationChanged="PhoneApplicationPage_OrientationChanged">
```

The `OrientationChanged` event handler receives an `OrientationChangedEventArgs` parameter which is used to determine the new orientation. The next listing shows how the event args are used to determine which visual state should be activated.

Listing 11.2 Switching states when orientation changes

```
private void PhoneApplicationPage_OrientationChanged(object sender,
    OrientationChangedEventArgs e)
{
    if ((e.Orientation & PageOrientation.Landscape)
        == PageOrientation.Landscape)
    {
        VisualStateManager.GoToState(this, "Landscape", true);
    }
    else if ((e.Orientation & PageOrientation.Portrait)
        == PageOrientation.Portrait)
    {
        VisualStateManager.GoToState(this, "Portrait", true);
    }
}
```

➊ **Bitwise operation**

The event handler gets the new orientation from the passed-in `OrientationChangedEventArgs` parameter, and uses the new orientation value to determine how the layout should be updated. When checking the `Orientation` property, a bitwise operation ➊ is performed to determine whether the orientation value is `Portrait` or `Landscape`. The layout is updated by calling the `GoToState` with the appropriate visual state name.

An application will always start in a portrait orientation and if the phone is currently held in a portrait orientation, the `OrientationChanged` event won't be fired. If the user is holding the phone sideways, the `OrientationChanged` event will be fired sometime after the `OnNavigatedTo` and `Loaded` events. Even when an application only supports landscape mode, the application will start in a portrait orientation and then switch to landscape before the `Loaded` event is called.

When returning from dormant, the page will be placed in a portrait orientation before the `OnNavigatedTo` method is called, even if the phone is currently in landscape mode. The `OrientationChanged` event will fire again to put the page back into landscape.

We've demonstrated how to use `VisualStates` and the `VisualStateManager` to implement different views of an application. In this sample, we used two different `VisualStates` to present one view for portrait orientation and another view for landscape orientation. You should consider using `VisualStates` in other situations, such as when you have unsaved edits, or an application is in offline mode.

We're finished with the OrientationWithStates sample application. In the remainder of the chapter we look at how to use features of Silverlight for Windows Phone and the control libraries to build user interfaces that conform to the Windows Phone Metro look and feel.

11.2 Building user interfaces

The Windows Phone operating system allows the user to choose a system-wide theme and accent color. The theme can be either light or dark, and the accent color can be one of nearly a dozen different colors. The Silverlight control library is designed to automatically work with the system theme. You should design your user interfaces to work with the system theme as well.

Silverlight for Windows Phone makes this easy by injecting a number of static resources into your application. The static resources are comprised of brushes, styles, font names, colors, and other types. You can view all of the available theme resources at http://mng.bz/HKT1.

Silverlight for Windows Phone has some unique performance constraints. The hardware platforms that run the operating system aren't very powerful compared to desktop computers that run Silverlight for the browser applications. Developers should keep performance a priority, and understand how to make efficient use of Silverlight controls.

You're going to see how to use the system resources with a few of the built-in controls. We'll look at how to bind styles and brushes to `TextBlock`, `Borders,` and shapes. We'll then examine the `ProgressBar` and `Image` controls, and learn how to efficiently use them within a Windows Phone application.

Throughout the remainder of this chapter we show you snippets of code but don't build a formal sample application. All of the snippets presented here have been combined into a single sample project, which is included with the sample source code available from the book's website.

11.2.1 TextBlock

The `TextBlock` control is the easiest control to use to display static text to the user and you've used the `TextBlock` control extensively in the sample applications for the book. A typical usage of the `TextBlock` control is to display a label for another control, or to display read-only messages. The `Text` property is used to get or set the text value display in the user interface:

```
textblock1.Text = "Hello World";
```

Like every Silverlight control, the `Text` property has an XAML counterpart:

```
<TextBlock Text="PhoneTextNormalStyle"
   Style="{StaticResource PhoneTextNormalStyle}" />
```

What's interesting in this snippet is how you use one of the theme style resources. Using the theme style gives you compatibility with the user experience guidelines, and is a good starting point to keep consistency between your application and the rest of the Windows Phone 7 ecosystem. Styles and other theme resources are applied via the `StaticResource` markup extension. Visuals Studio's XAML property editor can be used to select static resources. To select a theme resource for `TextBlock`'s `Style` property, click the Advanced Properties icon in the property editor (see figure 11.2).

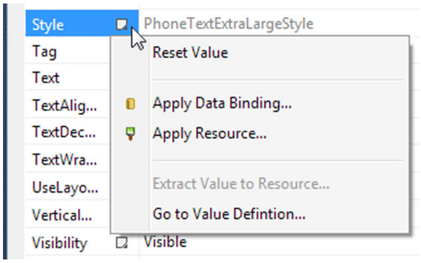


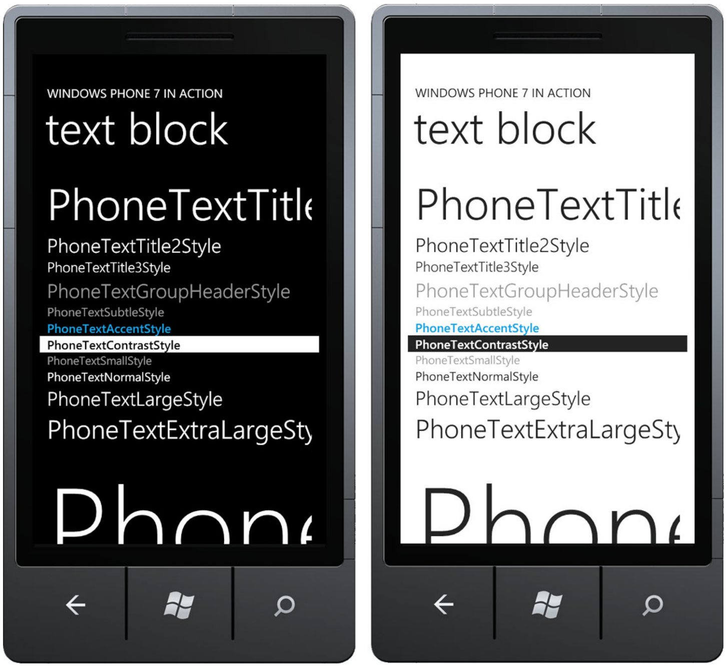
Figure 11.2 Advanced Properties context menu for a `TextBlock`'s Style property

Selecting the Apply Resource option from the context menu will bring up the resource picker. The resource picker allows you to search for resources defined in the framework and those defined within your application. The available theme text styles are

- `PhoneTextAccentStyle`
- `PhoneTextContrastStyle`
- `PhoneTextExtraLargeStyle`
- `PhoneTextGroupHeaderStyle`
- `PhoneTextHugeStyle`
- `PhoneTextLargeStyle`
- `PhoneTextNormalStyle`
- `PhoneTextSmallStyle`
- `PhoneTextSubtleStyle`
- `PhoneTextTitle1Style`
- `PhoneTextTitle2Style`
- `PhoneTextTitle3Style`

When you use the theme text styles, you're assured that your text will match the theme colors chosen by the user. Three different title styles, a group header style, and five different-sized text styles are defined. There are also styles defined for subtle, accent, and contrast text. Figure 11.3 displays the text styles in both the dark and light themes.

The `PhoneTextContrastStyle`'s `color` matches the theme's background, and text drawn with it won't be visible unless the background has been changed. In figure 11.3,



WINDOWS PHONE 7 IN ACTION

text block

PhoneTextTitle

PhoneTextTitle2Style

PhoneTextTitle3Style

PhoneTextGroupHeaderStyle

PhoneTextSubtleStyle

PhoneTextAccentStyle

PhoneTextContrastStyle

PhoneTextSmallStyle

PhoneTextNormalStyle

PhoneTextLargeStyle

PhoneTextExtraLargeSty

Phone

Figure 11.3 Theme text styles shown in the dark and light themes

the contrast text is placed inside a `Border` control whose background is drawn with the `PhoneContrastBackgroundBrush`.

11.2.2 Border

The `Border` control is used to generate a border frame, a background, or both, and is useful when generating a custom control. In practice, a border control is usually paired with other controls, as shown in figure 11.4.

To achieve the same effect, you need a `Border` with rounded corners containing a `TextBlock`:

```
<Border x:Name="border1" Height="100" Width="300" CornerRadius="30"
    BorderBrush="{StaticResource PhoneBorderBrush}"
    BorderThickness="{StaticResource PhoneBorderThickness}"
    Background="{StaticResource PhoneAccentBrush}}">
    <TextBlock x:Name="textBlock1" Text="Rounded TextBlock"
        VerticalAlignment="Center" HorizontalAlignment="Center" />
</Border>
```

The properties that are the most important for borders are `BorderThickness` and `CornerRadius`, which defines the radius for the corner. If `CornerRadius` is set to 0, then you get a `Rectangle`; otherwise you can tune the rounding effect on the corners.

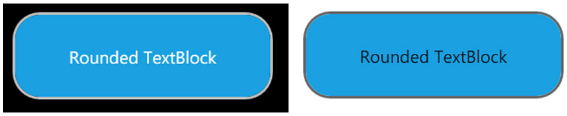


Figure 11.4 A `Border` with rounded corners using theme resources, shown in the dark and light themes

In this case you've changed the color of the background of the control to the theme accent color. You've also applied the theme's border color and thickness resources to the Border control. You don't need any special code to keep the accent color synchronized with the user settings.

Shapes can be used as an alternative to a `Border` control.

11.2.3 Shapes

The Silverlight `Shape` class serves as the base class for a variety of shape elements. Derived classes include `Line`, `Polyline`, `Polygon`, `Rectangle`, and `Ellipse`. Typical usage of these controls is to assemble them along with other components to generate the UI for a custom control. Shapes can be declared in XAML and can be bound to theme resources just like any other `FrameworkElement`:

```
<Ellipse x:Name="ellipse1" Margin="125,50,125,450"
    Stroke="{StaticResource PhoneContrastBackgroundBrush}"
    StrokeThickness="{StaticResource PhoneStrokeThickness}"
    Fill="{StaticResource PhoneSubtleBrush}" />
```

Here you're creating an `Ellipse` with its `Stroke` (border) brush set to `PhoneContrastBackgroundBrush`, and its `Fill` (background) brush set to `PhoneSubtleBrush`. The `StrokeThickness` is also bound to a theme resource, a `Thickness` object named `PhoneStrokeThickness`.

11.2.4 ProgressBar

When an application needs to perform some long-running task, it should provide the user some feedback that work is progressing while they wait. The `ProgressBar` control is one of the most common ways to convey to the user feedback that something is happening. The `ProgressBar` has two states—*indeterminate* and *determinate*. Indeterminate progress bars show an animation representing activity that will take an unknown amount of time, which will be stopped when the activity completes. Determinate progress bars are used when the number of steps or percentage complete is known, and the progress displays how many steps have been completed. A screenshot of the two states is shown in figure 11.5.

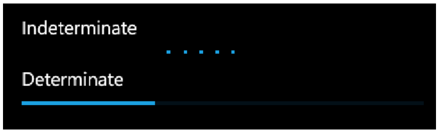


Figure 11.5 An Indeterminate progress bar (top) and a Determinate progress bar (bottom)

Let's see how an indeterminate `ProgressBar` works:

```
<ProgressBar x:Name="progressBar1" IsIndeterminate="True" />
```

The indeterminate state can be achieved by setting the `IsIndeterminate` property to true. This triggers the continuous animation, which can be stopped by setting the `IsIndeterminate` property to false.

TIP The indeterminate animation is resource-intensive. It's important to set the `IsIndeterminate` property to `false` instead of hiding the progress bar because the indeterminate progress animation continues to run when the control is collapsed.

The default value for `IsIndeterminate` is false, and the property doesn't need to be specified when declaring a determinate progress bar:

```
<ProgressBar x:Name="progressBar2" Value="35" Maximum="100" Minimum="0" />
```

The properties you likely want to set for determinate progress bars are as follows:

- `Value`—Represents the current progress to be shown by the progress bar
- `Minimum`—Value representing the minimum progress value
- `Maximum`—Value representing the maximum progress value

You'll update the `Value` property from your code as your long-running task completes its various steps.

Progress bars are often used when downloading images and data from the internet. Images can be easily downloaded and displayed with the `Image` control.

11.2.5 Image

It's easy to show images in Windows Phone 7 using the `Image` control. The image formats supported are JPEG and PNG. An image can be loaded directly by the `Image` control by specifying the filename, URI, or resource name for the `Source` property:

```
<Image x:Name="image1" Height="125" Width="185" Stretch="Fill"
    Source="/UserInterfaceControls;component/Images/cover_resource.png" />
```

Images can be compiled into your assembly by setting the image file's build action property to `Resource`. Resource images can be selected with the property editor using Visual Studio's image picker. The `Uri` specified here represents a resource-based image, and contains the assembly name and the path to the image file. Image resources are added to the assembly with a root path of *component*.

The manner in which the `Image` control fits the image depends on the `Stretch` property. Figure 11.6 demonstrates three different `Stretch` enumeration values. `Fill` will cause the image to be stretched to fill both the height and width of the control. `Uniform` will result in the image displayed in the correct proportions, with the entire image displayed. `UniformToFill` will cause a portion of the image to be cut off, with the image displayed in the correct proportions.

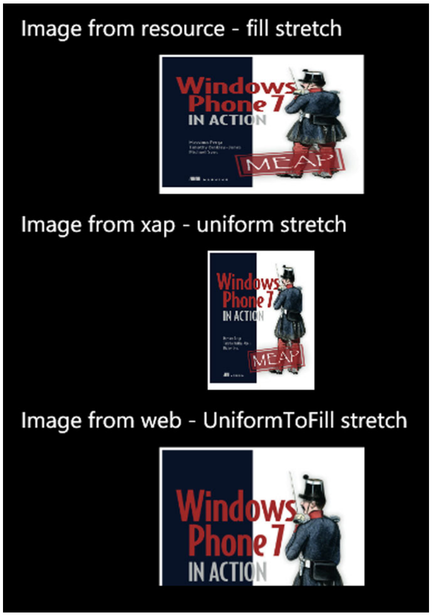


Image from resource - fill stretch
Windows Phone 7
IN ACTION
MEAP
Image from xap - uniform stretch
Windows Phone 7
IN ACTION
MEAP
Image from web - UniformToFill stretch
Windows Phone 7
IN ACTION

Figure 11.6 Images displayed with different `Stretch` values

Instead of embedding image files into the assembly, image files can be packaged in the .xap file by specifying a build action of `Content`. To improve assembly load time and media processing performance, it's recommended that you include images as `Content` instead of `Resources`. Content-based image files can be loaded by the `Image` control by specifying the path of the file relative to the root of the .xap file:

```
<Image x:Name="image2" Height="125" Width="185"
    Stretch="Uniform" Source="/Images/cover.png" />
```

Images can also be loaded from the internet by providing the full `Uri` of the image file:

```
<Image x:Name="image3" Height="125" Width="185" Stretch="UniformToFill"
    Source="http://www.wp7inaction.com/cover.png" />
```

The `Image` control will automatically download the image file and set it into the user interface. While the image is being downloaded, the Image control will be blank. You can also set the image's `Source` from code, but you need to create the `BitmapImage` object yourself:

```
image3.Source = new BitmapImage(
    new Uri("http://www.wp7inaction.com/cover.png",
    UriKind.Absolute));
```

The Windows Phone supports both JPG and PNG image formats. Image files must be smaller than 2000 x 2000 pixels. When choosing between the two supported image formats, JPG and PNG, you should consider the performance and characteristics of each format. If your image contains transparent pixels, use the PNG format. If your image doesn't contain any transparent pixels, choose the JPG format, because the JPG decoder provides better performance. This is especially important if you're displaying a large number of images in a list.

The `Image`, `TextBlock`, `Border`, and `ProgressBar` are just a few of the controls provided by the Windows Phone SDK that can be used to display data to the user. Let's take a look at a few controls whose function is to retrieve data from the user.

11.3 Receiving Input

Silverlight for Windows Phone provides the same basic user input controls that you might find in other user interface libraries, including a variety of buttons, a text box, and a slider. `Buttons` are used to trigger actions or to allow the user to set Boolean states. The `TextBox` control is used to get textual input typed in by the user. The `Slider` control allows the user to set constrained numeric values.

Each of the input controls behave slightly differently than you might expect, since the only interaction with the user is via the touch screen. The exception is when the user has a physical keyboard.

11.3.1 Button

You've used the `Button` control in sample applications in previous chapters. We'll focus here on generating the `Click` event. Silverlight for Windows Phone 7 can generate the `Click` event depending on the value specified in the `ClickMode` property. Just like WPF and Silverlight for the browser, the `ClickMode` property can have the values `Press`, `Release`, and `Hover`.

`Release` is the default mode, and raises the `Click` event when the user taps on the button and then releases the tap. If the user taps outside the button, drags over the button, and releases, no event is raised. If the user taps over the button, drags outside the button, and releases, no event is raised.

`Press` is similar to release, except that the event is raised as soon as the user taps on the button. If the user taps outside the button and then drags over the button, no event is raised.

The `Hover ClickMode` behaves differently than the other two modes. On a desktop computer, `Hover ClickMode` will raise an event as soon as the mouse is over the button. What happens on the phone where there isn't a mouse? `Hover` clicks can still be used on the phone. The click event is raised when the user taps outside the button and then drags over the button. The click event is also raised when the user taps and holds over the button. The click event will be repeatedly fired if the user drags off and back over the button in the same tap and hold gesture.

The `Button` control also converts touch gestures into mouse events such as `MouseLeftButtonDown`, `MouseLeftButtonUp`, and `MouseEnter`, although it's not recommended that you use the mouse events in your application. To improve performance, you should consider using the touch manipulation events instead.

11.3.2 HyperlinkButton

A `HyperlinkButton` inherits from the `Button` control and can be used in specific scenarios. As suggested by its name, The `HyperlinkButton` allows the user to navigate to another location using a hyperlink. The limitation is that the hyperlink must reference a page within your application. The `HyperlinkButton` won't automatically launch to a URI in a web browser. The property which contains the URI where you want the application execution to land is `NavigateUri`:

```
<HyperlinkButton x:Name="hyperlink1"
    Content="Hyperlink button sample"
    NavigateUri="/BuildingTheUI/TextBlockSample.xaml"
    Margin="{StaticResource PhoneTouchTargetOverhang}" />
```

In this snippet, you're setting the `Uri` to a page named TextBlockSample.xaml stored in a project folder named BuildingTheUI. You're also setting the `Padding` property to a theme resource named `PhoneTouchTargetOverhang`. You do this to prevent the hyperlinks from being placed too close together and to increase the touch target. Fingertips aren't as precise as mouse pointers, and a larger touch target improves the user experience.

The `NavigateUri` can be set from code as well:

```
hyperlink1.NavigateUri =
    new Uri("/BuildingTheUI/TextBlockSample.xaml", UriKind.Relative);
```

If you do want to create a hyperlink to a web page, you should implement a `Click` event handler that uses the `WebBrowserTask` to launch Internet Explorer. You could also create a page within your application that hosts the `WebBrowser` control, and construct the `NavigateUri` with a query string parameter containing the `Uri` that the web browser should load. The `WebBrowserTask` and the `WebBrowser` control will be covered in chapter 13. Constructing and parsing navigation URIs with query string parameters was covered in chapter 2.

11.3.3 CheckBox

The `CheckBox` control is a button that can have two or three states. The three-states mode can be enabled using the `IsThreeState` property. Figure 11.7 demonstrates the three visual states for a `CheckBox`.

You can get or set the `CheckBox`'s state using the `IsChecked` property. `IsChecked` is a nullable Boolean. Normally, a Boolean (or any other value type) must contain a value. Nullable types allow a value type to be assigned a null value. When working with the `IsChecked` property in code, you should allow for a null value. Even when a `CheckBox` isn't in three-state mode, `IsChecked` can be set to null in XAML or code-behind:

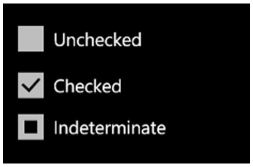


Figure 11.7 Three possible states for the `CheckBox` control. Indeterminate is available only when IsThreeState is set to true.

```
if (!sampleCheckBox.IsChecked.HasValue)
    Debug.WriteLine("sampleCheckBox is indeterminate");
else if (sampleCheckBox.IsChecked.Value)
    Debug.WriteLine("sampleCheckBox is checked");
else
    Debug.WriteLine("sampleCheckBox is unchecked");
```

In this snippet, you first check whether the `IsChecked` property has a value. If `HasValue` is false, you know the `CheckBox` is indeterminate; otherwise you can use the Boolean true or false value.

11.3.4 RadioButton

The `RadioButton` is similar to a check box, with the difference that two or more radio buttons must be grouped together. The property to be used to group two or more radio buttons is `GroupName`:

```
<StackPanel>
    <RadioButton x:Name="radioButton1" Content="option 1"
        GroupName="myGroup" IsChecked="True" />
    <RadioButton x:Name="radioButton2" Content="option 2"
         GroupName="myGroup" />
    <RadioButton x:Name="radioButton3" Content="option 3"
        GroupName="myGroup" />
</StackPanel>
```

The `RadioButton` and `CheckBox` controls both inherit their `IsChecked` property from the `ToggleButton` base class. This means that `RadioButton`'s `IsChecked` property is also a nullable Boolean. Even though the user can't place a `RadioButton` into an indeterminate state, be aware that the `IsChecked` property could potentially have a null value if it's set from code behind or XAML. To determine which radio button in the group is checked, you must check each of them:

```
if (radioButton1.IsChecked.Value)
    Debug.WriteLine("option 1 selected");
else if (radioButton2.IsChecked.Value)
    Debug.WriteLine("option 2 selected");
else if (radioButton3.IsChecked.Value)
    Debug.WriteLine("option 3 selected");
```

Having to check each control is a bit onerous, but there's no actual object that represents the group that can be queried to determine which option is selected.

`Buttons`, `HyperLinks`, `CheckBoxes`, and `RadioButtons` are all controls that the user interacts with by tapping. Let's look at controls where the user actually enters data.

11.3.5 TextBox

Text boxes, along with text blocks, are common controls you find in most every user interface. A `TextBox` allows the user to insert data and Windows Phone supports single- and multi-line editing (see figure 11.8). The `Text` property is used to set or get its content, and its type is `string`.

The Boolean property used to determine whether a `TextBox` is acting in single-line is `AcceptsReturn`. When set to `true`, `AcceptsReturn` enables multi-line input; otherwise only single-line input is accepted.

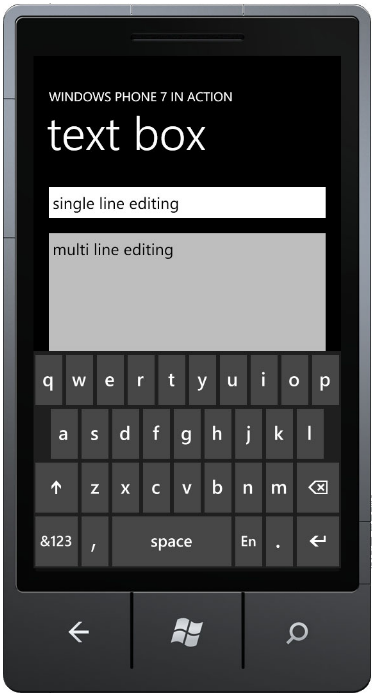


Figure 11.8 A single line `TextBox`, a multi-line `TextBox`, and the software input panel

Let's see an example where you create two `TextBox` controls, the first one for the single-line editing and the second one with multi-line editing:

```
<StackPanel>
    <TextBox x:Name="singlelineTextBox" Text="single line editing" />
    <TextBox x:Name="multilineTextBox" Height="255"
        Text="multi line editing"
        AcceptsReturn="True" TextWrapping="Wrap" />
</StackPanel>
```

Multi-line `TextBoxes` may contain text that won't fit on the screen. The `TextBox` has built-in support for displaying scroll bars. A horizontal scroll bar can be displayed using the `HorizontalScrollBarVisibility` property. `VerticalScrollBarVisibility` controls the display of a vertical scroll bar. Both the properties accept a value defined in the `ScrollBarVisibility` enumeration.

Other useful properties supported by the `TextBox` are `TextWrapping`, `MaxLength`, and `InputScope`. When text wrapping is enabled, long lines will be wrapped onto multiple lines, even if they don't contain a linefeed character. The `MaxLength` property limits the maximum number of characters that can be entered into the `TextBox`. The `InputScope` property can be used to control which software input panel is displayed to the user.

CONTROLLING THE KEYBOARD WITH INPUT SCOPE

Windows Phone 7 devices have been designed to be fully touch-enabled and a physical keyboard isn't present on many models. When a `TextBox` has the focus on a device that doesn't have a physical keyboard in use, a virtual keyboard or software input panel (SIP) is displayed. By default, the `TextBox` invokes a SIP with the standard QWERTY keyboard layout and doesn't provide text correction or suggestions. A number of different keyboard layouts are available, each one tailored to a specific task. Input scope names are defined by the `InputScopeNameValue` enumeration. Table 11.1 lists a few of the input scope names that are useful to Windows Phone applications.

Table 11.1 Common input scopes

Input scope name	Description
`Chat`	A QWERTY keyboard layout with text suggestions and an emoticons key. Automatic correction isn't enabled.
`EmailNameOrAddress` `EmailSmtpAddress`	A QWERTY keyboard with @ and .com keys. Text suggestions and correction aren't enabled.
`Maps`	A QWERTY keyboard layout with text suggestions and a customized enter key. Automatic correction isn't enabled.
`NameOrPhoneNumber`	A QWERTY keyboard layout with a semicolon key and access to the 12-key number pad. Text suggestions and correction aren't enabled.
`TelephoneNumber`	12-key number pad with * and #. See figure 11.9.
`Search`	A QWERTY keyboard layout with a customized enter key. Text suggestions and automatic correction aren't enabled.
`Text`	A QWERTY keyboard layout with text suggestions and automatic correction enabled.
`Url`	A QWERTY keyboard layout with a .com key and a customized enter key. Text suggestions and automatic correction aren't enabled.

This is only a partial list of input scopes. For a full list, see the MSDN documentation for the `InputScopeNameValue` enumeration.

When developing your application, you should choose the most appropriate keyboard layout for your `TextBox` controls. If your `TextBox` accepts standard text phrases, you should consider using one of the layouts that provide suggestions and/or automatic correction. If your application automatically performs work or navigation once the user has input some text, you might want to use one of the layouts that provide a customized

Enter key, which suggests some action will occur when Enter is pressed. If you have an input field that only accepts numbers, you might consider using the `TelephoneNumber` input scope (see figure 11.9).

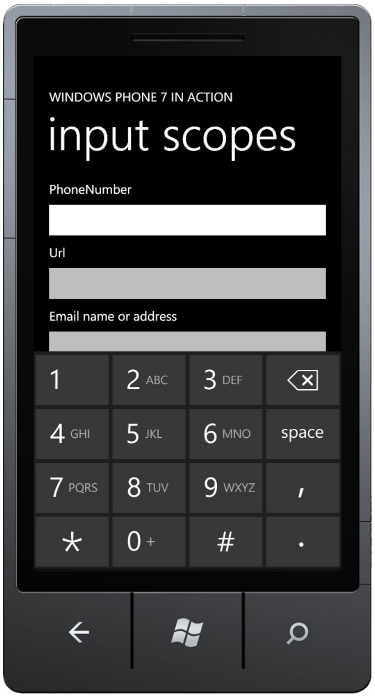


Figure 11.9 The 12-key layout displayed when using the `TelephoneNumber` input scope

Setting the input scope for a specific text box is easy and can be accomplished both programmatically by code and at design time with XAML. The `InputScope` and `InputScopeName` classes are found in the `System.Windows.Input` namespace:

```
InputScope virtualKeyboard = new InputScope();
InputScopeName keyboardLayout = new
     InputScopeName();
keyboardLayout.NameValue =
     InputScopeNameValue.TelephoneNumber;
virtualKeyboard.Names.Add(keyboardLayout);
textBox1.InputScope = virtualKeyboard;
```

To set the input scope in code, you start by creating an instance of `InputScope` and an instance of `InputScopeName`. You set the `InputScopeName`'s `NameValue` property to `TelephoneNumber`. Note that `InputScopeNameValue` is an enumeration containing all the defined input scopes. Finally, you add the keyboard layout to the `InputScope` and assign it the `TextBox`'s `InputScope` property.

Setting the input scope in XAML is much simpler. You only need to enter the input scope name for the `InputScope` property. The XAML engine will perform the appropriate conversion from string to an `InputScope` instance:

```
<TextBox x:Name="textBox1" InputScope="TelephoneNumber"/>
```

More than 60 different values are defined in the `InputScopeNameValue` enumeration. See the MSDN documentation for the full list and how each one affects the `TextBox`.

The `TextBox` is ideal for entering text, but not so nice for working with numbers. The `Slider` control is more suited for incrementing and decrementing numeric values.

11.3.6 Slider

The `Slider` control is used to represent a well-known range of numeric values allowed to be entered by the user. For example, with the `Slider` you can restrict the possible values of the slider to a range from 0 to 50 and a minimum variation in the slider range of 5:

```
<Slider x:Name="slider1" LargeChange="5" Value="15" Minimum="0"
    Maximum="50" />
<TextBlock x:Name="slider1Value" HorizontalAlignment="Center"
   Text="{Binding ElementName=slider1, Path=Value}" />
```

In this snippet, you added a `TextBlock` to display the current value of the slider. You bound the `Content` of a `TextBlock` control to the `Value` of the `Slider` control.

The relevant properties for `Slider` control are `Value`, `Minimum`, `Maximum`, and `LargeChange`. The `Value` property represents the current value of the control. The `Minimum` property is the value representing the smallest value that can be selected by the user. The largest value that can be selected by the user is declared in the `Maximum` property. The `LargeChange` property controls how much the value will change when the user touches the slider control.

> **NOTE** The `Slider` has a `SmallChange` property, but its values aren't used by the Windows Phone `Slider` control.

The `Slider` control is one of many controls that can be used to interact with the user. We haven't exhaustively covered each and every control, but we've looked at buttons and the text box. You've learned how these controls differ from Silverlight for the browser, and we showed you some things you should consider when using controls in a Windows Phone application.

11.4 Silverlight Toolkit for Windows Phone

Earlier in the chapter we introduced many of the standard controls that come with the Windows Phone SDK. Microsoft also distributes the Silverlight Toolkit for Windows Phone, which is another set of user interface components that mimic controls seen in the native Windows Phone 7 user interface but not provided with the SDK.

A few of the components available in the toolkit are as follows:

- `ContextMenu`
- `DatePicker`
- `ListPicker`
- `LongListSelector`
- `TimePicker`
- `ToggleSwitch`
- `WrapPanel`

The toolkit is available for free from CodePlex (http://silverlight.codeplex.com) and can be downloaded as a Windows installer. You can also download a zip file that includes the full source code along with a sample application. The Silverlight Toolkit installer will copy the toolkit artifacts to %Program Files%\Microsoft SDKs\Windows Phone\v7.1\Toolkit\Oct11\Bin.

> **NOTE** The final installation folder depends on the version of the Silverlight Toolkit you download. At the time of this writing, the current version was November 2011.

The Silverlight Toolkit is packaged in a single assembly named Microsoft.Phone.Controls.Toolkit.dll. You'll need to add a reference to this assembly before using any of the toolkit components. When working with toolkit components in XAML you'll need to include an XML namespace declaration:

```
xmlns:toolkit="clr-namespace:Microsoft.Phone.Controls;
➥ assembly=Microsoft.Phone.Controls.Toolkit"
```

In addition to user interface controls, the Silverlight Toolkit contains classes for working with gestures and performing page transitions. We can't cover the entire toolkit in this book, but we're going to take a closer look at the `ToggleSwitch`, `DatePicker`, `TimePicker`, `ContextMenu`, and `GestureListener`.

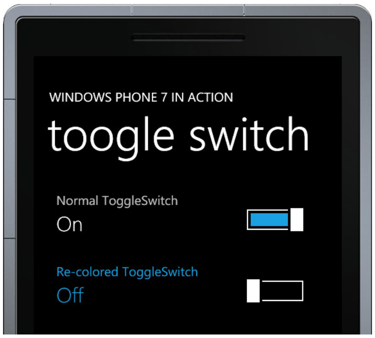
11.4.1 ToggleSwitch

The `ToggleSwitch` control is a component used to represent a choice to a user and is similar to the `CheckBox` and `RadioButton` controls, which derive from `ToggleButton`. The `ToggleSwitch` has a Boolean `IsChecked` property and raises `Click`, `Check`, `Indeterminate`, and `Uncheck` events. Even though `ToggleSwitch` has an interface similar to `ToggleButton`, it doesn't derive from `ToggleButton`.

One place where `ToggleSwitch` differs from `ToggleButton` is the lack of the `IsThreeState` property. The `ToggleSwitch.IsChecked` property can be set to a `null` value, and the `Indeterminate` event will be raised, but the user interface will look as if the `IsChecked` value is false.

Another difference between `ToggleButton` and `ToggleSwitch` is how the `Content` property is used. In a `ToggleButton`, the `Content` property is used for label text. In `ToggleSwitch`, the `Content` property is used as another way to render the `IsChecked` property. When the `IsChecked` property is `true`, the `Content` property displays the word *On*. As shown in figure 11.10, the word *Off* is displayed when `IsChecked` is `false` or `null`.

If the `Content` property is used to display On/Off, you might be wondering how you specify label text for a `ToggleSwitch`. The `ToggleSwitch` exposes the `Header` property to add label text to a `ToggleSwitch`:


WINDOWS PHONE 7 IN ACTION

toogle switch

Normal ToggleSwitch

On

Re-colored ToggleSwitch

Off

Figure 11.10 **`ToggleSwitch` control provided by the Silverlight Toolkit for Windows Phone 7**

```
<toolkit:ToggleSwitch Header="Normal ToggleSwitch" />
```

Though you might not be able to distinguish the colors in figure 11.10, the switch's fill color is normally the color specified in the `PhoneAccentBrush` resource. The label text in the `Header` is normally rendered using the `PhoneBorderBrush` theme resource. The `PhoneForegroundBrush` theme resource is used to render the On/Off text in the `Content`.

The `ToggleSwitch` provides the `SwitchForeground` property to allow the developer to specify a new color. The `Foreground` property controls the color of the On/Off text. Changing the `Header` text color isn't so easy.

The `ToggleSwitch` `Header` property can be customized by declaring a new `HeaderTemplate`. The following listing shows how to use the `Foreground`, `SwitchForeground`, and `HeaderTemplate` properties to recolor a `ToggleSwitch`.

Listing 11.3 Customizing colors of the `ToggleSwitch` control

```
<toolkit:ToggleSwitch Header="Re-colored ToggleSwitch"
        SwitchForeground="{StaticResource PhoneChromeBrush}"
        Foreground="{StaticResource PhoneAccentBrush}">
    <toolkit:ToggleSwitch.HeaderTemplate>
        <DataTemplate>
            <ContentControl Content="{Binding}"
                Foreground="{StaticResource PhoneAccentBrush}" />
        </DataTemplate>
    </toolkit:ToggleSwitch.HeaderTemplate>
</toolkit:ToggleSwitch>
```

❶ **Change color**

❷ **New template**

In this listing, you declare the recolored `ToggleSwitch` and set the `Header` property to your label text. You then use a couple of theme resource brushes ❶ to set the `SwitchForeground` and `Foreground` properties. Finally, you declare a simple `DataTemplate` ❷ with a single `ContentControl`, setting the `Foreground` property of the control to the same theme resource you use for the `Foreground` property of the `ToggleSwitch`.

The toggle switches are used to display on/off status. They're used throughout the native applications and especially when configuring settings. Date and time pickers are a couple of controls used in native applications and are also not part of the Windows Phone SDK.

11.4.2 DatePicker and TimePicker

The native Windows Phone calendar application uses a couple of unique controls for picking data and time. The Silverlight Toolkit provides managed implementations of these pickers with the `DatePicker` and the `TimePicker` controls. These two picker controls, shown in figure 11.11, are both composed of a button that displays the current value and a secondary page with scrolling selectors.

The phone page shown on the left of figure 11.11 is built by placing both a `DatePicker` and a `TimePicker` inside a `StackPanel` container. The label for each control is set with the `Header` property:

```
<StackPanel>
    <toolkit:DatePicker Header="Date picker" Value="05/24/2011" />
    <toolkit:TimePicker Header="Time picker" Value="09:45 AM" />
</StackPanel>
```

Both picker controls expose a `DateTime` property named `Value`. When you declare dates or times in XAML, the string is converted using the `TimeTypeConverter` class, which expects an English format. The pickers raise a `ValueChanged` event when the `Value` property is changed.

There are a couple of caveats when using the `DatePicker` and `TimePicker` controls. The first caveat centers around the icons that appear on the application bar of

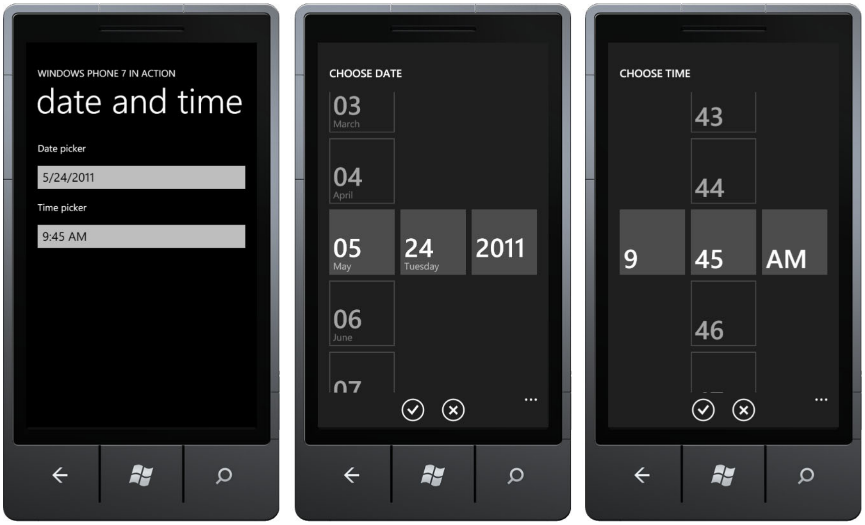


Figure 11.11 Date and time pickers provided by the Silverlight Toolkit

the secondary picker pages. The application bar requires icons to be in files shipped in the application's .xap file. This means you need to create a folder in your project named Toolkit.Content with two files named ApplicationBar.Cancel.png and ApplicationBar.Check.png respectively for the cancel and check icons. You can provide your own icons files or you can use the files included in the Silverlight Toolkit. The toolkit icons can be copied from %Program Files%\Microsoft SDKs\Windows Phone\v7.1\Toolkit\Oct11\Bin\Icons. After you add the icons files to your project, be sure to set their build action to `Content`.

The second warning stems from how the picker controls display their secondary selector pages. When the user taps the picker, the control navigates to the secondary page. This means the page hosting the picker control is pushed onto the navigation stack, and the `OnNavigatedFrom` method override method will be called. When the user returns from the selector page, the host page's `OnNavigatedTo` method override will be called.

With the `DatePicker` and `TimePicker` controls, you can add features to your application so that it'll behave like the native phone applications. Native applications also use context menus to expose features to the user.

11.4.3 ContextMenu

In the last chapter you learned how to create menu items on the application bar. Though this is a good place to put access to application level features, sometimes you need a menu specific to a single object in the user interface. The Silverlight Toolkit provides a context menu implementation you can use for these situations.

The `ContextMenu`, and the related `ContextMenuService`, provide the toolkit implementation of a context menu. A context menu is displayed when the user performs a

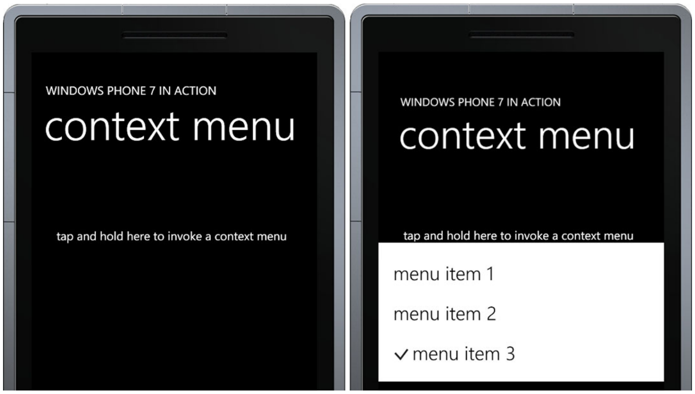

WINDOWS PHONE 7 IN ACTION
context menu
tap and hold here to invoke a context menu
WINDOWS PHONE 7 IN ACTION
context menu
tap and hold here to invoke a context menu
menu item 1
menu item 2
✓ menu item 3

Figure 11.12 A context menu demonstrating the zoom effect and a complex menu item header

tap and hold gesture on a user interface component. When the context menu is shown, it'll shrink the page providing the appearance that the menu pops out of the application. This zoom effect, shown in figure 11.12, is optional and is controlled by using the `IsZoomEnabled` property.

The `ContextMenuService` class provides the dependency properties that allow a `ContextMenu` to be declared in XAML using attached property syntax. The `ContextMenu` can be attached to any user interface element. The next listing demonstrates how to attach a `ContextMenu` to a `TextBlock`.

Listing 11.4 Adding a `ContextMenu` to a `TextBlock`

```
<TextBlock Margin="30,100,30,100"
        Text="tap and hold here to invoke a context menu">
    <toolkit:ContextMenuService.ContextMenu>
        <toolkit:ContextMenu>
            <toolkit:MenuItem x:Name="menuItem1" Header="menu item 1"
                Click="MenuItem_Click" />
            <toolkit:MenuItem x:Name="menuItem2" Header="menu item 2"
                Click="MenuItem_Click" />
        </toolkit:ContextMenu>
   </toolkit:ContextMenuService.ContextMenu>
</TextBlock>
```

❶ **Attached property**

❷ **Menu text declared with Header**

First you add a `ContextMenu` using attached property syntax ❶. Next you add two `MenuItems`, providing each with a name and using the `Header` property to declare the menu text ❷. You wire up the `Click` event of both `MenuItems` to the same event handler, a method named `MenuItem_Click`:

```
private void MenuItem_Click(object sender, RoutedEventArgs e)
{
    var menuItem = (MenuItem)sender;
    MessageBox.Show(menuItem.Name, "Menu Item Clicked",
        MessageBoxButton.OK);
}
```

The click event handler casts the `sender` property to a `MenuItem`. A message box is shown to inform the user that the menu item was clicked. Individual menu items can be enabled or disabled at runtime using their `IsEnabled` properties.

The display of a `MenuItem` can be customized by declaring custom markup for the `Header`, or by creating a `HeaderTemplate`. In the following listing, you create a `MenuItem` that displays a checkmark next to the menu item text.

Listing 11.5 A `MenuItem` with complex header content

```
<toolkit:MenuItem x:Name="menuItem3" Click="MenuItem_Click">
    <toolkit:MenuItem.Header>
        <StackPanel Orientation="Horizontal">
            <Path Margin="0,0,6,0" Data="M 3 23 L 12 33 L 24 17"
                Stroke="{StaticResource PhoneChromeBrush}"
                StrokeThickness="{StaticResource PhoneStrokeThickness}" />
            <TextBlock Text="menu item 3" />
        </StackPanel>
    </toolkit:MenuItem.Header>
</toolkit:MenuItem>
```

❶ **Property element syntax**

You declare the `Header` using property element syntax ❶. The `Header` content is a horizontal `StackPanel` containing a `Path` and a `TextBlock`. The `Path` draws the shape of a checkmark and uses theme resources for the `Stroke` and `StrokeThickness` properties.

The `ContextMenu` is invoked in response to the user performing a tap and hold gesture. Tap and hold is just one type of several gestures that can be used by a user to interact with an application. The `GestureListener` class allows your application to listen for touch gestures.

11.4.4 GestureListener

All of the built-in user interface controls are designed to respond to touch gestures. Buttons respond to tap gestures. List boxes and slider controls scroll when detecting a drag or flick gesture. When building your application, you may choose to provide customized responses to touch gestures. Silverlight offers `Touch.FrameReported`, `ManipulationStarted`, `ManipulationDelta`, and `ManipulationCompleted` events for detecting touches, but none of these APIs offer convenient gesture support. The `TouchPanel` class in the XNA Framework provides the `ReadGesture` API, but its polling model isn't suited for event-based Silverlight applications.

The `GestureListener` class wraps both the `Touch.FrameReported` and `TouchPanel` APIs with a convenient eventing interface. `GestureListener` supports tap, double tap, drag, flick, pinch, and hold gestures. The events raised by `GestureListener` are listed in Table 11.2.

Each of the `GestureListener` events provides details about the gesture through an `EventArgs` class derived from `GestureEventArgs`. The screen coordinate of the touch related to the gesture can be obtained from the `GestureEventArgs` `GetPosition` method. Each of the `GestureEventArgs`-derived classes provide details specific to the type of

Table 11.2 Events raised by the GestureListener

Gesture	Event Args	Description
`DoubleTap*`	`GestureEventArgs`	Two quick touch and release motions in quick succession.
`DragComplete`	`DragCompletedGestureEventArgs`	A drag motion has ended.
`DragDelta`	`DragDeltaGestureEventArgs`	The touch point has moved during a drag gesture.
`DragStarted`	`DragStartedGestureEventArgs`	A drag motion has started.
`Flick`	`FlickGestureEventArgs`	A touch followed by a quick swiping motion.
`GestureBegin`	`GestureEventArgs`	A gesture has started.
`GestureCompleted`	`GestureEventArgs`	An ongoing gesture has ended.
`Hold*`	`GestureEventArgs`	A touch that is not immediately released.
`PinchCompleted`	`PinchGestureEventArgs`	A pinch motion has ended
`PinchDelta`	`PinchGestureEventArgs`	A touch point has moved during a pinch operation.
`PinchStarted`	`PinchStartedGestureEventArgs`	Two touches followed by movement bringing the two points closer together or farther apart.
`Tap*`	`GestureEventArgs`	A quick touch and release.

* Starting with Windows Phone SDK 7.1, `Tap`, `DoubleTap`, and `Hold` events have been added to the `UIElement` class and you don't need to use `GestureListener` to detect tap, double tap, and hold gestures.

gesture they represent. You can see an example of the gesture data in figure 11.13. We leave it as an exercise for the reader to take a closer look at the `EventArgs` classes.

The `GestureService` class provides the dependency properties that allow a `GestureListener` to be declared in XAML using attached property syntax. The `GestureListener` can be attached to any user interface element. The next listing demonstrates how to attach a `GestureListener` to a `Rectangle`.

Listing 11.6 Attaching a `GestureListener`

```
<Rectangle Margin="50" Grid.Row="1"
        Fill="{StaticResource PhoneAccentBrush}">
    <toolkit:GestureService.GestureListener>
        <toolkit:GestureListener Tap="gesture_Tap"
            DoubleTap="gesture_DoubleTap"
            Hold="gesture_Hold"
            Flick="gesture_Flick"
            DragStarted="gesture_DragStarted"
```

① **Attached property**

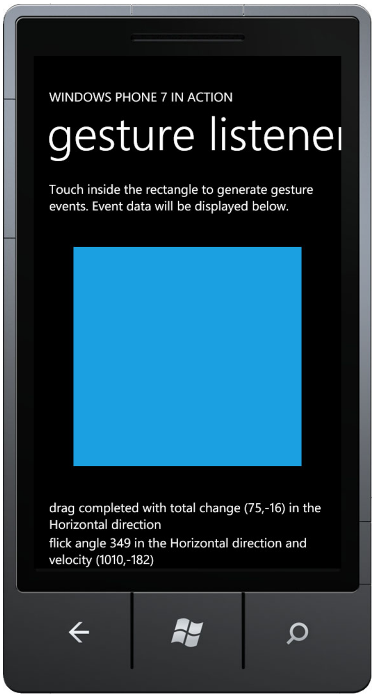
② **Event handlers**

```
            DragDelta="gesture_DragDelta"
            DragCompleted="gesture_DragCompleted"
            PinchStarted="gesture_PinchStarted"
            PinchDelta="gesture_PinchDelta"
            PinchCompleted="gesture_PinchCompleted" />
    </toolkit:GestureService.GestureListener>
</Rectangle>
```

② **Event handlers**

First you add a `GestureListener` using attached property syntax ①. Next you wire the gesture events. In this example you're subscribing to most of the gesture events ②. In your applications, you should only listen to the gestures that are required by your implementation. You should be aware that some of the gestures overlap and you may receive multiple events from the same touch operation, such as flick and drag.

The `GestureListener`, `ToggleSwitch`, `DatePicker`, `TimePicker`, and `ContextMenu` are only a few of the components available in the Silverlight Toolkit for Windows Phone. We recommend that you visit the Silverlight Toolkit home on CodePlex for more information on the entire library. The website for the toolkit project is http://silverlight.codeplex.com.



WINDOWS PHONE 7 IN ACTION

gesture listener

Touch inside the rectangle to generate gesture events. Event data will be displayed below.

drag completed with total change (75,-16) in the Horizontal direction

flick angle 349 in the Horizontal direction and velocity (1010,-182)

Figure 11.13 `GestureListener` sample displaying event data

11.5 Summary

From its very beginning, Silverlight was designed as a platform that would run on multiple different platforms. This portability makes Silverlight an ideal technology for building user interfaces on the Windows Phone. In this chapter we looked at several Silverlight components found in the Windows Phone SDK and the Silverlight Toolkit. You learned how they're similar to components found in Silverlight for the browser.

The Windows Phone isn't a browser, and some features of the controls library are unique to the phone. You learned how theme resources are injected into a running application as static resources, and how they can be used to ensure an application matches the system look and feel. Since a physical keyboard may not be available, you learned how to interact with the software input panel provided by the operating system. Finally, we looked at how touch gestures can be captured and used by an application.

We started our look at Silverlight controls in the last chapter, discussing the `ApplicationBar`, `Pivot`, and `Panorama` controls. We're not finished looking at the controls available to the phone developer. In the next chapter we use the `MediaElement` control that's part of the Windows Phone SDK and we look at the `SmoothStreamingMediaElement`, which is part of the Smooth Streaming SDK open source. We wrap up our coverage of Silverlight controls in chapter 13 with a close look at the Bing Maps and `WebBrowser` controls.

12

Manipulating and creating media with MediaElement

This chapter covers

- Building a media player
- Working with local and web media
- Creating custom media streams
- Streaming video

One of the more popular uses of a mobile phone is as a media player. Developers have a variety of different options when building applications that play media. In chapter 7, we showed you how to integrate with the Music + Videos Hub and to make use of the XNA Framework and the `BackgroundAudioPlayer` to play audio. In this chapter we're going to look at two Silverlight controls that can be used to play both audio and video from inside an application.

These controls are `MediaElement`, which plays downloaded media files, and `SmoothStreamingMediaElement`, which incrementally downloads media files as they're being played. With these two controls you can get a full-featured media player with a few lines of code. In this chapter you'll see how to implement a basic media player in Silverlight for Windows Phone using the `MediaElement` control. With this example you'll learn how to render different media types supported by Windows Phone and control the playback. The media containers supported by

Windows Phone through the `MediaElement` control include WAV, MP3, WMA, 3GP, 3G2, MP4, M4A, WMV, and M4V.

We'll then move to a lower level, where we'll explore how to dynamically generate video and audio content and get them rendered using custom `MediaStreamSource` implementations. Multimedia is usually delivered using a container, which works like an envelope, as well as content. When working with media players, there are times when the software is able to work with the container format but not with the content and vice versa. `MediaStreamSource` capability provided by `MediaElement` is useful not only with dynamically generated media, but when you have to deal with container formats that aren't supported by `MediaElement` but whose contents are supported by Windows Phone decoders.

In this chapter you're going to build two variations of a simple media player application. Figure 12.1 shows screenshots of the application and the different features you'll build. We'll demonstrate how to play media that's distributed in the .xap file as part of the application. We'll also discuss how to play media that's located on the internet, or has been saved into the application's isolated storage.

This sample application will also show you how to implement your own video and audio streams. You'll build two different `MediaStreamSource` implementations and hook them up to your media player. You'll use Silverlight's `MediaElement` control to load and display your media.

The second sample application will be similar to the first and will reuse most of the XAML and code behind. You'll replace the `MediaElement` control with a `SmoothStreamingMediaElement` from the IIS Smooth Streaming Client library. IIS Smooth

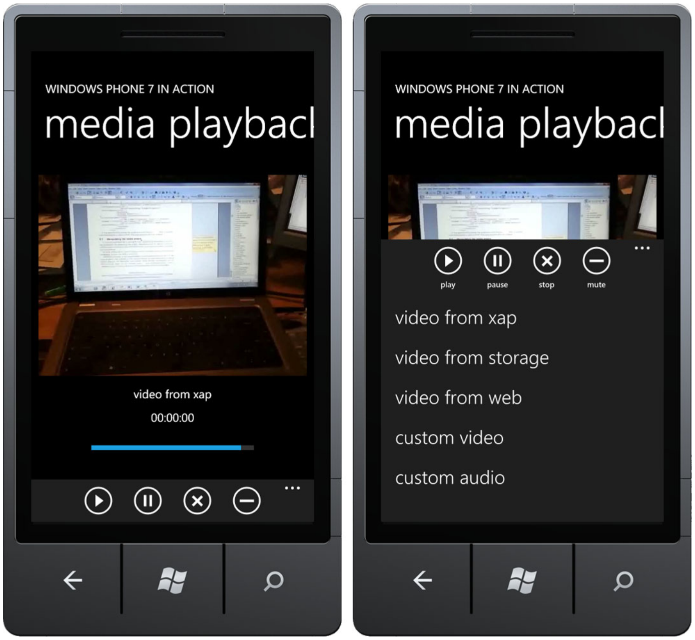


Figure 12.1 The Media Playback sample application

Streaming is Microsoft's technology for streaming media. Smooth Streaming incrementally downloads media while it's being played. Smooth Streaming adapts the quality of the media to match the performance and bandwidth of the playback device and the network connection.

These media playback controls allow you to build custom media players for your audio and video content. Let's see how easy this is by starting with the `MediaElement` control.

12.1 Building a media player with MediaElement

`MediaElement` is a user interface control that got its start in the Windows Presentation Foundation (WPF) framework and has been part of the Silverlight framework since version 2. `MediaElement` for Silverlight isn't as full-featured as the WPF implementation, and the Windows Phone edition is even more limited. The media containers supported by Windows Phone through the `MediaElement` control include WAV, MP3, WMA, 3GP, 3G2, MP4, M4A, WMV, and M4V.

> **TIP** Some of the codecs supported by a real device aren't supported by the emulator. For the full list of containers and decoders supported by Windows Phone and the emulator, see the official Microsoft documentation at http://msdn.microsoft.com/en-us/library/ff462087(VS.92).aspx.

`MediaElement` itself isn't what many users would call a media player. It's just a surface for displaying video. `MediaElement` doesn't provide any user interface to control playback such as VCR-like buttons for play, pause, and stop. `MediaElement` does provide methods and properties to enable you to build your own media player user interface.

In this section you'll learn how to open and play media files with `MediaElement`. We'll show you how to play files locally and from the internet. Finally we'll show you how to implement common player controls and how to control volume. Let's jump right into code by showing you how to create a media player application.

12.1.1 Creating the media player project

Before we discuss how to use `MediaElement` to play audio and video files, you need to build the skeleton user interface for your application. Start by creating a new project using the Windows Phone Application project template, and name the project MediaPlayback. The application's screen displays a `MediaElement`, along with a few status controls hosted inside the `ContentPanel`. The following listing shows the starting XAML for the `ContentPanel`.

Listing 12.1 User interface skeleton

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="40" />
        <RowDefinition Height="40" />
        <RowDefinition Height="80" />
```

❶ **Define four rows**

```
    </
     Grid.RowDefinitions>

    <MediaElement x:Name="mediaElement" Source="Media/sample.wmv" />

    <TextBlock x:Name="sourceTextBlock" Text="video from xap" +++
       TextAlignment="Center" Grid.Row="1" />
</Grid>
```

Media player ❷

Display media source ❸

You divide the `ContentPanel` into four rows, allowing the first row to use up all the available space ❶. A `MediaElement` is defined to live in the first grid row, and defaults to using a video deployed as content in the Media folder of the application's .xap file ❷. You also add a `TextBlock` to the second row ❸ to display the selected source option. You'll add controls to the other rows as you work through the chapter.

You now need to add playback controls for the play, pause, stop, and mute operations. The playback controls will be placed on the `ApplicationBar`. The different media source options are listed on the `ApplicationBar`'s menu. The XAML for the `ApplicationBar` is shown in the next listing.

Listing 12.2 MediaPlayback's `ApplicationBar`

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
        <shell:ApplicationBarIconButton Text="play"
            Click="PlayClicked"
            IconUri="/Images/appbar.transport.play.rest.png" />
        <shell:ApplicationBarIconButton Text="pause"
            Click="PauseClicked"
            IconUri="/Images/appbar.stop.rest.png" />
        <shell:ApplicationBarIconButton Text="stop"
            Click="StopClicked"
            IconUri="/Images/appbar.transport.pause.rest.png" />
        <shell:ApplicationBarIconButton Text="mute"
            Click="MuteClicked"
            IconUri="/Images/appbar.minus.rest.png" />
        <shell:ApplicationBar.MenuItems>
            <shell:ApplicationBarMenuItem Text="video from xap"
                Click="VideoFromXapClicked" />
            <shell:ApplicationBarMenuItem Text="video from storage"
                Click="VideoFromStorageClicked" />
            <shell:ApplicationBarMenuItem Text="video from web"
                Click="VideoFromWebClicked" />
            <shell:ApplicationBarMenuItem Text="custom video"
                Click="CustomVideoClicked" />
            <shell:ApplicationBarMenuItem Text="custom audio"
                Click="CustomAudioClicked" />
        </shell:ApplicationBar.MenuItems>
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

❶ **Playback buttons**

❷ **Location options**

❸ **Custom media options**

`ApplicationBarIconButton`s are used to control playback and `ApplicationBarMenuItem`s are used to load media content from different locations and sources. Following the style guide, the button text is a single lowercase word ❶. The .xap, storage, and web options will demonstrate how to load media from a .xap file, a web URL, or a file that has been saved in isolated storage ❷. The custom audio and video options will be used to load custom `MediaStreamSources` that you'll implement later in the chapter ❸. You'll implement the menu item click event handlers in the next few sections.

TIP The images used in this sample are available in the Windows Phone SDK.

Let's see how to implement the click event handlers for the buttons. You can start a playback using the `Play` method exposed by `MediaElement` control:

```
private void PlayClicked(object sender, RoutedEventArgs e)
{
    mediaElement.Play();
}
```

In a similar way, you can use the `Pause` and `Stop` methods to control playback:

```
private void PauseClicked(object sender, RoutedEventArgs e)
{
    mediaElement.Pause();
}

private void StopClicked(object sender, RoutedEventArgs e)
{
    mediaElement.Stop();
}
```

You've just seen how to control the playback. Now let's see how to show the playback progress. You can use the `Position` property, which will show the playback time in terms of hours, minutes, and seconds. You can present the current progress by binding the `MediaElement`'s `Position` property to the `TextBlock`'s `Text` property:

```
<TextBlock Name="positionTextBlock"
    Text="{Binding ElementName=mediaElement, Path=Position}"
    TextAlignment="Center" Grid.Row="2" />
```

The `positionTextBlock` is added to the third row of the `ContentPanel`, right below the `TextBlock` that displays the current source option.

TIP When you display the value of the `Position` property in your user interface, you should be aware that the `Position` property can change hundreds of times per second. In a production application, you should avoid data binding the property, and implement another mechanism that updates the user interface less frequently.

The sample video you're including in the .xap file and launching when the application starts is a WMV file. You could've chosen a file in any of the supported formats and your code would be exactly the same (except the filename). The `MediaElement` discovers the format when it loads the media. You're loading sample.wmv by specifying a relative URI when declaring the `MediaElement` control in XAML, which is one of several methods that can be used to load media files.

12.1.2 Loading media files

`MediaElement` can load files using relative or absolute URIs, or can play media directly from an `IsolatedStorageFileStream`. Relative `URIs`, such as Media/sample.wmv used in the previous section, refer to files (and folders) deployed in the application's .xap file. Absolute URIs are used to load media files located on the internet. Media files can also be stored in isolated storage.

Media files are placed into the .xap file by adding the file to the project. Once you've added the media file, you can set the build action property to either `Resource` or `Content`. If you choose `Resource`, the file will be embedded into the assembly. When a file is embedded into an assembly, this will increase the application's startup time, so only use this option for small media files. If you choose `Content`, the file will be added to the .xap file.

In either case, the file increases the size of the deployment package. Any package larger than 20 MB will prevent the user from downloading the application from the Windows Phone Marketplace with the cellular network and users will only be able to download your application when connected to a Wi-Fi or Ethernet network. Application packages must not exceed a total of 225 MB.

You instruct `MediaElement` to load a file from a `URI` by setting the `Source` property. When you set the `Source` property, `MediaElement` locates the file and opens the corresponding file stream. You'll use the `Source` property to implement the `VideoFromXapClicked` event handler for the Video from the xap menu option:

```
private void VideoFromXapClicked(object sender, EventArgs e)
{
    mediaElement.Source = new Uri("Media/sample.wmv", UriKind.Relative);
    sourceTextBlock.Text = "video from xap";
}
```

In addition to setting the source element to a relative `Uri`, you update `sourceTextBlock` to display which option was selected. Loading a file from the internet is nearly the same:

```
private void VideoFromWebClicked(object sender, EventArgs e)
{
    mediaElement.Source = new Uri(
        "http://www.wp7inaction.com/sample.wmv", UriKind.Absolute);
    sourceTextBlock.Text = "video from web";
}
```

The main differences between `VideoFromXapClicked` and `VideoFromWebClicked` are the `Uri` and `UriKind` used for the `Source` property, and the text used to update `sourceTextBlock`. `MediaElement` will download the specified media file before starting playback. The download process might take some time, and the `MediaElement` doesn't provide any indication that it's waiting for the download. In the next section we'll discuss how you can use `MediaElement` events to determine when a file is fully loaded from a `Uri`.

Loading media files from isolated storage isn't as simple as specifying a `Uri` for the `Source` property. You must use `MediaElement`'s `SetSource` method instead, as shown in the following listing. Instead of using a `Uri`, `SetSource` takes a `Stream`.

Listing 12.3 Loading media files from isolated storage

```
private void VideoFromStorageClicked(object sender, EventArgs e)
{
    mediaElement.Source = null;
    using (var store = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (store.FileExists("sample.wmv"))
        {
            var fileStream = new IsolatedStorageFileStream(
                "sample.wmv", FileMode.Open, store);
            mediaElement.SetSource(fileStream);
            sourceTextBlock.Text = "video from storage";
        }
    }
}
```

Close current media

① **Open media file**

② **Pass stream to mediaElement**

In the `VideoFromStorageClicked` event handler, you start by opening isolated storage and confirming that the target media file already exists. Next you open an `IsolatedStorageFileStream` for the file ①. Finally, you pass the stream to the `MediaElement` ②.

> **NOTE** The `MediaElement` in Silverlight for Windows Phone only supports `IsolatedStorageFileStreams`.

When an application is first deployed to a device, isolated storage is empty, and there's no mechanism for prepopulating files. There are three options for placing files into isolated storage—create new files, copy files from the .xap file, or download files from the network. Because this application requires a file in isolated storage, you're going to add code required to copy the embedded sample.wmv file from the .xap file to isolated storage. You want to add this to the code that's run when the application is launched. The next listing details the code added to `Application_Launching` event handler in App.Xaml.cs.

Listing 12.4 Copying content files into isolated storage

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    using (var store = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (!store.FileExists("sample.wmv"))
        {
            using (var stream = Application.GetResourceStream(
                new Uri("Media/sample.wmv", UriKind.Relative)).Stream)
            using (var fileStream = new IsolatedStorageFileStream(
                "sample.wmv", FileMode.Create, store))
            {
                byte[] bytesInStream = new byte[stream.Length];
```

① **Only copy file once**

② **Open resource stream**

③ **Create file stream**

```
                stream.Read(bytesInStream, 0, (int)bytesInStream.Length);
                fileStream.Write(bytesInStream, 0, bytesInStream.Length);
                fileStream.Flush();
            }
        }
    }
}
```

You start by opening isolated storage and checking whether the file has already been copied ❶. You then use the `GetResourceStream` method to open a `Stream` ❷ for the sample.wmv file stored in a folder name Media in the .xap file. You also create a new `IsolatedStorageFileStream` ❸. Finally, you copy the bytes from one `Stream` to the other.

> **TIP** Though not shown here, a similar method can be used for saving files from the internet into isolated storage. Instead of using the stream returned by `GetResourceStream`, you use the `Stream` returned in the `WebClient.OpenReadCompleted` event handler.

One more `MediaElement` property is important when the `source` is set or changed. The `AutoPlay` property determines whether media playback is automatically started when the source changes. `AutoPlay` defaults to true, meaning the media will begin playing as soon as it's opened by the `MediaElement`. A good practice is to explicitly set the `AutoPlay` before setting the `Source` property to ensure the `MediaElement` behaves as you expect.

We've shown you how to use various methods to load media files. Loading files isn't instantaneous, especially when loading files from the internet. When `MediaElement` is loading a file, it doesn't provide any progress or wait indicator to the user. Fortunately, `MediaElement` reports its current status so that you can build your own progress indicators.

12.1.3 Media element states

`MediaElement` contains a number of different properties and events that can be used to determine the current status of media playback and exert control over the user experience. The `MediaOpened` event can be used to determine when the `MediaElement` is ready to start playing. An application can be notified when media has stopped playing with the `MediaEnded` event. The `BufferingProgress` or `DownloadProgress` properties, and their related changed events, can be used to identify situations when you might display a wait indicator to the user.

Many of `MediaElement`'s events and properties are fairly low-level and require the programmer to track various properties and events in order to track when media is playing, and when it's stopped or paused. Fortunately, `MediaElement` exposes the `CurrentState` property so that user interface code can easily determine the playback state. The `CurrentState` property is of type `MediaElementState` and the possible values are detailed in table 12.1.

You're going to use the `CurrentState` property to update the user interface with some helpful media playback hints. You're going to display the global progress indicator

Table 12.1 MediaElement states

State*	Description
`Buffering`	Media frames are being loaded and prepped for playback. During buffering, the `Position` property doesn't change, and if the media type is video, the current frame continues to be displayed.
`Closed`	The media source hasn't been set, or has been cleared.
`Opening`	The media stream is being downloaded and opened and the media type discovered as the `MediaElement` prepares to play.
`Paused`	The currently playing media is paused, and the `Position` property doesn't change. If the media type is video, the current frame continues to be displayed.
`Playing`	Media is being played and the `Position` property is changing.
`Stopped`	Media is loaded, but isn't being played. The `Position` property isn't changing, and has the value 0. If the media type is video the first frame is displayed.

*Two other states exist, `AcquiringLicense` and `Individualizing`, which apply to digital rights management and are beyond the scope of this book.

during media opening so the user sees feedback when the media file takes a while to load, such as when opening a file from the internet. You add the `ProgressIndicator` to MainPage.xaml, but initialize it in an invisible state so that it's hidden from the user. Add the following code below the application bar markup in MainPage.xaml:

```
<shell:SystemTray.ProgressIndicator>
    <shell:ProgressIndicator x:Name="mediaProgress" IsIndeterminate="True"
        IsVisible="True" Text="Loading..."/>
</shell:SystemTray.ProgressIndicator>
```

You also set the `IsIndeterminate` property to `True` because you're not tracking actual download size or detailed opening progress. You'll add an event handler that will display the `ProgressIndicator` when the `MediaElement` is opening a media file. First, add a `TextBlock` to display the value of the `CurrentState`:

```
<TextBlock x:Name="stateTextBlock" Grid.Row="1"
    HorizontalAlignment="Right" />
```

The `TextBlock` will be displayed in the same row as and to the right of the `TextBlock` that displays the current source option. You want to update the `TextBlock` whenever the `CurrentState` property changes, or in the `CurrentStateChanged` event handler. Now wire up the `CurrentStateChanged` event to the `MediaElement`:

```
<MediaElement x:Name="mediaElement" ...
    CurrentStateChanged="mediaElement_CurrentStateChanged" />
```

You implement `mediaElement_CurrentStateChanged` in MainPage.xaml.cs:

```
private void mediaElement_CurrentStateChanged(
    object sender, RoutedEventArgs e)
{
    stateTextBlock.Text = mediaElement.CurrentState.ToString();
    if (mediaElement.CurrentState == MediaElementState.Opening)
        mediaProgress.IsVisibile = true;
    else
        mediaProgress.IsVisibile = false;
}
```

You update the `stateTextBlock` with the value of the `CurrentState`. Because you want to display the `ProgressIndicator` only when the media source is being opened, you check for the `Opening` state and make the `ProgressIndicator` visible. For all other states, you hide the `ProgressIndicator`.

What happens if there's an error during load or playback? There's no `MediaElementState` for error. Instead you're going to handle the `MediaFailed` event and report an error to the user. You wire up the `MediaFailed` event in MainPage.xaml:

```
<MediaElement x:Name="mediaElement" ...
    MediaFailed="mediaElement_MediaFailed" />
```

You implement the `mediaElement_MediaFailed` event handler in MainPage.xaml.cs:

```
private void mediaElement_MediaFailed(
    object sender, ExceptionRoutedEventArgs e)
{
    MessageBox.Show(e.ErrorException.Message,
        "Media Failure", MessageBoxButton.OK);
}
```

The error message from the `Exception` is displayed in a `MessageBox`. The `MediaFailed` event represents a generic failure, so it can be related either to the file location or to media content not properly understood by the `MediaElement`. Usually, this message is just an error code, so you may consider inspecting the exception and building a more user-friendly message in your code.

You now know how to load media files and have looked at some of the methods and properties provided by `MediaElement` to control playback. One of the features we haven't looked at yet is controlling sound volume.

12.1.4 Controlling volume

Your media player application would be incomplete without controls to adjust the volume or mute all sound. The `MediaElement` control provides the Volume and `IsMuted` properties to support these desired features. The `Volume` property is represented by a `double` value in the range between 0 (silent) and 1 (maximum volume). You're going to implement the volume adjustment user interface with a `Slider` control. You can then configure the `Slider` control to have a range between 0 and 1, with resolution for changes of 0.1:

```
<Slider x:Name="volumeSlider" Width="300" Grid.Row="3" Minimum="0.0"
    Maximum="1.0" SmallChange="0.05" LargeChange="0.1" Value="0.85"/>
```

You initialize the `Slider` control's `Value` property to 0.85, which is the default value for the `MediaElement`'s `Volume` property. You add the `Slider` to the fourth row of the

`ContentPanel` grid. Finally, you use data binding to connect the `Slider`'s `Value` to the `MediaElement` `Volume` property:

```
<MediaElement x:Name="mediaElement" ...
    Volume="{Binding ElementName=volumeSlider, Path=Value}" />
```

The user can now silence the media player by dragging the slider all the way to zero. A quicker method for silencing the media player is with the `IsMuted` property. You can instantly silence the media player by setting the `IsMuted` Boolean property to true. When you created the application, you added a mute button to the `ApplicationBar`. Now you'll implement the `Click` event handler of the mute button to toggle the `IsMuted` property:

```
private void muteButton_Click(object sender, RoutedEventArgs e)
{
    mediaElement.IsMuted = !mediaElement.IsMuted;
    mutedTextBlock.Text = mediaElement.IsMuted ? "muted" : string.Empty;
}
```

In the event handler, you're also updating a `TextBlock` to provide the user with feedback when the media player is muted. You haven't declared the `mutedTextBlock` yet, so add a new `TextBlock` control to MainPage.xaml:

```
<TextBlock x:Name="mutedTextBlock" Grid.Row="3"
    HorizontalAlignment="Right" VerticalAlignment="Bottom" />
```

You add the `TextBlock` to the bottom row of the `ContentPanel`, to the right of the volume control. When the media player is muted, the control displays the word *muted*, and displays nothing otherwise.

Though not quite as powerful and feature-rich as its WPF and Silverlight for the browser cousins, the `MediaElement` in Silverlight for Windows Phone should meet most of your media playback needs. Of course no control can be expected to fulfill every need of every developer. When you find yourself with a problem that can't be solved by `MediaElement`'s built-in features, you can extend the control with custom `MediaStreamSource` implementations.

12.2 Manipulating the media stream with MediaStreamSource

In the previous section we covered the basic capabilities of the `MediaElement` control. `MediaElement` and its relatives are the only method allowed to reproduce audio and video content in an application written in Silverlight for Windows Phone. What if you want to produce sounds or videos dynamically without using preexisting resources like audio or video files?

`MediaElement` embeds a powerful feature named `MediaStreamSource` that solves these problems. For example, `MediaElement` may fail to play media (and raise the `MediaFailed` event) whose decoder is supported by the Windows Phone device but whose container isn't. Using `MediaStreamSource` you can implement your own container

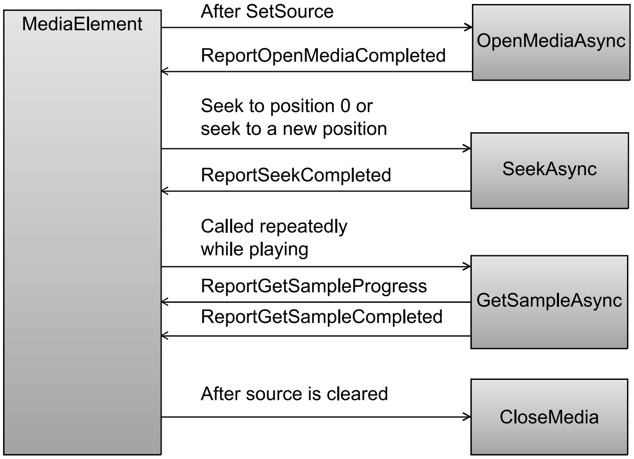

MediaElement
After SetSource
ReportOpenMediaCompleted
OpenMediaAsync
Seek to position 0 or
seek to a new position
ReportSeekCompleted
SeekAsync
Called repeatedly
while playing
ReportGetSampleProgress
ReportGetSampleCompleted
GetSampleAsync
After source is cleared
CloseMedia

Figure 12.2 `MediaStreamSource` lifecycle

and provide the media samples directly to the decoders. Obviously such a powerful feature requires more work by the developer.

`MediaStreamSource` deals with media samples and not directly with files. All the code required to open files or network streams must be coded by the developer. If you decide to opt for a `MediaStreamSource`-based solution, you need to create a class that inherits from `MediaStreamSource`. `MediaStreamSource` is an abstract class and has several abstract methods that must be implemented by your derived class.

Figure 12.2 shows the typical cycle the `MediaStreamSource` goes through for each media stream. Methods shown in figure 12.2 are some of those left for you to implement.

`MediaElement` uses an asynchronous pattern to communicate with a stream source. When `MediaElement` calls to one of the `XxxAsync` methods, it doesn't wait for the method to complete before moving on to other work. When the async method has completed its work, it must report its results to the `MediaElement` through an appropriate `ReportXxxCompleted` method.

12.2.1 Opening a media source

After a stream source is properly constructed and set into the `MediaElement`, the first method invoked is `OpenMediaAsync`. This is the method to use when opening the media source. `OpenMediaAsync` should construct two collections. The first is a dictionary mapping `MediaSourceAttributesKeys` to string values. The second is a collection of `MediaStreamDescriptions`.

The media source attributes describe features of the media source. In Silverlight, this collection is limited to `Duration`, `CanSeek`, and `DRMHeader` attributes. The `Duration` attribute should be set to the number of nanoseconds of playback time for the media stream. If the `Duration` is unknown, set the attribute's value to zero. The `CanSeek` attribute should be set to `True` or `False`, indicating whether or not the media stream can be positioned to play at any random point. Working with the `DRMHeader` and general digital rights management is beyond the scope of this book.

The following code snippet demonstrates how to create and populate the dictionary:

```
var attr = new Dictionary<MediaSourceAttributesKeys, string>();
attr[MediaSourceAttributesKeys.Duration] = "0";
attr[MediaSourceAttributesKeys.CanSeek] = "False";
```

A `MediaStreamSource` may contain multiple media streams. The `MediaStreamDescription` class describes each of the streams contained in the media source. The description specifies whether the stream is audio or video, provides an identifier, and also contains a dictionary of `MediaStreamAttributes`. Stream attributes include the height and width of the video, as well as which codec is used to process the stream. The following listing shows how a collection of `MediaStreamDescriptions` might be created.

Listing 12.5 Building a collection of `MediaStreamDescriptions`

```
var vAttr = new Dictionary<MediaStreamAttributeKeys, string>();
vAttr[MediaStreamAttributeKeys.VideoFourCC] = "RGBA";
vAttr[MediaStreamAttributeKeys.Height] = "320";
vAttr[MediaStreamAttributeKeys.Width] = "480";

var aAttr = new Dictionary<MediaStreamAttributeKeys, string>();
aAttr[MediaStreamAttributeKeys.CodecPrivateData] = codecData;

var streams = new List<MediaStreamDescription>();
streams.Add( new MediaStreamDescription(MediaStreamType.Video, vAttr));
streams.Add( new MediaStreamDescription(MediaStreamType.Audio, aAttr));
```

① **Codec-related data**

Depending on the type of media stream you're working with, you may need to use the `CodecPrivateData` attribute ①. If you're working with audio, this is an encoded `WaveFormatEx` structure, and we'll show you how to generate wave data later in the chapter. Video streams may require codec-specific data which is beyond the scope of this book.

Once the source attributes and stream descriptions have been created, you send the data to the `MediaElement` control by calling the `ReportOpenMediaCompleted` method. `OpenMediaAsync` is called once during the whole media session. Opening media is just the first step in the process `MediaElement` uses as it prepares to play media. The next step is to position the playback position to the beginning of the media stream.

12.2.2 Seeking media

Once the media source is opened, `MediaElement` will ask the media source to position its media stream to position zero. This will occur even when the source `CanSeek` attribute is false. `MediaElement` will also call `SeekAsync` with a position of zero when `MediaElement.Stop` has been called, followed by a call to `MediaElement.Play`.

If your media supports seeking or repositioning the current location within the stream, the `CanSeek` source attribute should be set to `True`. When the `MediaElement`'s `Position` property is set from code, `MediaElement` calls the `SeekAsync` method.

When your implementation of `SeekAsync` is called, it must perform the necessary operations to position the stream at the requested location. When your `MediaStreamSource` implementation has prepared the stream, it must call the `ReportSeekCompleted` method to notify the `MediaElement` your application has properly updated the position. Now that the media stream is opened and its position set, how does `MediaElement` obtain the media samples it needs to play?

12.2.3 Sampling media

Once the `MediaStreamSource` is open and positioned at the beginning of the stream, `MediaElement` will start asking the media source for samples by calling `GetSampleAsync`. `GetSampleAsync` defines a `MediaStreamType` parameter that specifies the type of media sample to provide. The `MediaStreamType` enumeration contains `Audio` and `Video` values. Within `GetSampleAsync` you need to instantiate a `MediaStreamSample` object, which will contain the media description, a `MemoryStream` containing the actual sample, a dictionary containing `MediaSampleAttributeKeys`, and a timestamp. The next listing demonstrates how a `MediaStreamSample` might be built.

Listing 12.6 Building a `MediaStreamSample`

```
var attr = new Dictionary<MediaSampleAttributeKeys, string>();
attr[MediaSampleAttributeKeys.FrameHeight] = "320";
attr[MediaSampleAttributeKeys.FrameWidth] = "480";
attr[MediaSampleAttributeKeys.KeyFrameFlag] = "True";

var stream = new MemoryStream();
stream.Write(...);

var sample = new MediaStreamSample( description, stream, 0,
    stream.Length, timestamp, attr);
```

① Fill stream with data

② Create media sample

You start by creating and filling the dictionary with the height and width of the sample as well as setting a value for `KeyFrameFlag` attribute. Next you create a new `MemoryStream`. We've omitted the code where the memory stream would be filled with data ①. Finally, you create a new `MediaStreamSample` passing in the `MediaStreamDescription` instance ② that was created in `OpenMediaAsync`, the new stream, a timestamp, and the sample attributes. We've also omitted code that tracks and increments the timestamp.

Once done with the production of a sample, you can pass it back to the `MediaElement` to be rendered through the `ReportGetSampleCompleted` method. If you can't immediately return a sample, such as when you're buffering data from the network, you should periodically call `ReportGetSampleProgress` while you're waiting. Once the sample is ready, call `ReportGetSampleCompleted` and return.

You can inform the `MediaElement` about the end of a stream by creating a `MediaStreamSample` with a null memory stream.

`MediaElement` can ask an implementation of `MediaStreamSource` about buffering progress via the `GetDiagnosticAsync` method to implement. Buffering data is returned to the `MediaElement` by calling `ReportGetDiagnosticCompleted`. Media stream buffering is beyond the scope of this book.

Now that you know about building a custom `MediaStreamSource`, you're going to build a couple of your own. You're going to build an audio stream source that plays a steady tone that could be used for generating individual musical notes. The first sample will start with a custom video stream source.

12.3 Creating custom video

You're going to build a simple implementation of `MediaStreamSource` that provides video samples for one minute. These video samples will alternate once a second between blue video content and orange video content. When you're finished with your custom stream source, you'll display the generated video in the `MediaPlayback` sample application.

Start by creating a new class named `VideoMediaStreamSource` that inherits from `MediaStreamSource`. You need to define several fields, shown in the following listing, that will be used throughout the class.

Listing 12.7 Fields necessary to implement `VideoMediaStreamSource`

```
public class VideoMediaStreamSource : MediaStreamSource
{
    private const int BytesPerPixel = 4;
    private readonly int _frameWidth;
    private readonly int _frameHeight;
    private readonly int _framePixelCount;
    private readonly int _frameBufferSize;
    private readonly int _frameRate;

    private int _currentBufferFrame = 0;
    private int _currentReadyFrame = 1;
    private int _currentVideoTimeStamp = 0;

    private int _samplesProvided = 0;

    private byte[][] _frames = new byte[2][];

    private MediaStreamDescription _videoDesc;
    private readonly Dictionary<MediaSampleAttributeKeys, string>
        _sampleAttr;
}
```

❶ **Frame size fields**

❷ **Current frame fields**

❸ **Media stream fields**

First, you define a constant named `BytesPerPixel`, and since you're using an RGBA format that uses 32 bits, you set its value to 4. Next you declare some fields that you use to store frame parameters ❶. You also declare `_currentBufferFrame` to track the buffer index used for the current frame, and in `_currentReadyFrame` you store the frame ready to be shown ❷. You declare `_currentVideoTimeStamp`, which will be used to provide the right timestamp to the sample you dynamically generate. You then define the buffers that will store the two frames, which will flip every second.

You add a field to hold the `MediaStreamDescription` ❸ created in `OpenMediaAsync` and used to create `MediaStreamSamples`. Finally, you create a `Dictionary` to store the media sample attributes used when producing the media samples.

Now that you've defined the basic structure of the `VideoMediaStreamSource` class, you need to initialize the fields to appropriate values.

12.3.1 *Initializing the stream source*

You must initialize the read-only fields in the class constructor. Your stream source implementation allows different-sized video streams to be generated. The width and height of the video are passed in the constructor. The `VideoMediaStreamSource` constructor is shown in the next listing.

Listing 12.8 The `VideoMediaStreamSource` class constructor

```
public VideoMediaStreamSource(int frameWidth, int frameHeight)
{
    _frameWidth = frameWidth;
    _frameHeight = frameHeight;
    _framePixelCount = frameWidth * frameHeight;
    _frameBufferSize = _framePixelCount * BytesPerPixel;

    _frameRate = (int)TimeSpan.FromSeconds((double)1 / 50).Ticks;

    _frames[0] = new byte[_frameBufferSize];
    _frames[1] = new byte[_frameBufferSize];

    _sampleAttr = new Dictionary<MediaSampleAttributeKeys, string>();
    _sampleAttr[MediaSampleAttributeKeys.FrameHeight]
        = _frameHeight.ToString();
    _sampleAttr[MediaSampleAttributeKeys.FrameWidth]
        = _frameWidth.ToString();
    _sampleAttr[MediaSampleAttributeKeys.KeyFrameFlag] = "True";

    FillFrame(0, Colors.Orange);
    FillFrame(1, Colors.Blue);
}
```

① **Calculate frame size**

50 Hz frame rate

② **Allocate buffers**

③ **Create orange and blue frames**

First you make simple calculations about the frame based on the height and width passed to the constructor ①. You allocate the memory for one blue and one orange frame ②. Because you always have the same kind of media sample, you initialize the `_sampleAttr` dictionary with the frame size attributes. You complete the constructor by creating two frames filled completely with orange and blue pixels ③. The constructor uses the `FillFrame` method to fill each of the four byte pixels in the frame buffer:

```
public void FillFrame(int index, Color color)
{
    for (int i = 0; i < _framePixelCount; i++)
    {
        int offset = i * BytesPerPixel;

        _frames[index][offset++] = color.B;
        _frames[index][offset++] = color.G;
        _frames[index][offset++] = color.R;
        _frames[index][offset++] = color.A;
    }
}
```

To specify which frame to fill, you pass a value that represents the index into the `_frames` array. As the frame is an array of bytes, each pixel is spread across four positions in the array. Incrementing the offset value by one will have the effect of indexing the byte for the next color component. The bytes in the frame are filled in the order blue, green, red, and the alpha transparency value.

All this construction code will be called prior to adding the stream source to the `MediaElement`. When `MediaElement` gets a hold of an instance of your class, the first thing it does is try to open the stream.

12.3.2 Opening the video stream source

The next step in your implementation of `VideoMediaStreamSource` is the `OpenMediaAsync` method. As we mentioned earlier, `OpenMediaAsync` must build a dictionary of `MediaSourceAttributesKeys`, and a collection of `MediaStreamDescription` objects. The following listing demonstrates how these collections are created by your implementation.

Listing 12.9 Implementation of `OpenMediaAsync`

```
protected override void OpenMediaAsync()
{
    Dictionary<MediaSourceAttributesKeys, string> attr =
        new Dictionary<MediaSourceAttributesKeys, string>();
    List<MediaStreamDescription> availableStreams =
        new List<MediaStreamDescription>();

    PrepareVideo();

    availableStreams.Add(_videoDesc);

    attr[MediaSourceAttributesKeys.Duration] = "0"
    attr[MediaSourceAttributesKeys.CanSeek] = "False"

    ReportOpenMediaCompleted( attr, availableStreams);
}
```

❶ Create collections

❷ Add attributes

❸ Report complete

You start your implementation by instantiating the collections that will store the source attributes and the media stream description ❶. Next, you call the `PrepareVideo` method, which initializes the `_videoDesc` field that's added to the `availableStreams` collection. In this case, you'll have just one stream because you have only a video stream and aren't implementing any audio streams. You set the `Duration` attribute to infinite and disable the seek capability by setting the `CanSeek` attribute to `false` ❷. You complete the implementation by sending the newly built collections to the `MediaElement` by way of the `ReportOpenMediaCompleted` method ❸.

The internal `PrepareVideo` method creates a `MemoryStream` and populates a `MediaStreamDescription` instance with stream attributes. `PrepareVideo` is implemented in the next listing.

Listing 12.10 Preparing the media stream

```
private void PrepareVideo()
{
    Dictionary<MediaStreamAttributeKeys, string> attr =
        new Dictionary<MediaStreamAttributeKeys, string>();
    attr[MediaStreamAttributeKeys.VideoFourCC] = "RGBA";
    attr[MediaStreamAttributeKeys.Height] = _frameHeight.ToString();
    attr[MediaStreamAttributeKeys.Width] = _frameWidth.ToString();

    MediaStreamDescription msd = new MediaStreamDescription(
        MediaStreamType.Video, attr);
    _videoDesc = msd;
}
```

Set attributes ①

Create stream description ②

First you define the stream in terms of the attributes ① for color channels RGBA and frame size. Finally, you create a `MediaStreamDescription` instance where you declare that your media stream is a video with the attributes contained in the `attr` object ②, and you set it to the `_videoDesc` field.

When `OpenMediaAsync` returns, the video is ready to play. The `MediaElement` control will begin calling your class to obtain the video samples to draw to the screen. The next step is to generate the samples for the `MediaElement` to display.

12.3.3 Generating media samples

When `MediaElement` is ready to display your video stream, the control will call `GetSampleAsync`. Your orange and blue media samples will be generated on the fly in `GetSampleAsync`, which is shown in the following listing.

Listing 12.11 Generating the video samples

```
protected override void GetSampleAsync(MediaStreamType mediaStreamType)
{
    if (mediaStreamType == MediaStreamType.Video)
        throw new NotSupportedException();

    MemoryStream frameStream = new MemoryStream();
    frameStream.Write(_frames[_currentReadyFrame], 0,
         _frameBufferSize);

    MediaStreamSample msSamp = new MediaStreamSample(
        _videoDesc, frameStream, 0, _frameBufferSize,
        _currentVideoTimeStamp, _sampleAttr);

    _currentVideoTimeStamp += _frameRate;
    if ((_currentVideoTimeStamp % 10000000) == 0)
    {
        int f = _currentBufferFrame;
        _currentBufferFrame = _currentReadyFrame;
        _currentReadyFrame = f;
        _samplesProvided++;
    }
```

① Create new stream

② Create sample

③ Change colors

```
    if (_samplesProvided < 60)
    {
        ReportGetSampleCompleted(msSamp);
    }
    else
    {
        MediaStreamSample nullSample = new MediaStreamSample(
            _videoDesc, null, 0, 0, _currentVideoTimeStamp,
            _sampleAttr);
        ReportGetSampleCompleted(nullSample);
     }
}
```

❹ **Report sample**

❺ **Report end of stream**

You first check the stream type whose media sample is being requested by the `MediaElement`. Because you only support video, you throw a `NotSupportedException` if an audio sample is requested. When the requested `MediaStreamType` is `Video`, you instantiate a new `MemoryStream` for the frame and copy into it the buffer you've previously prepared ❶. Note that you're using `_currentReadyFrame` as the index to the target sample.

Next you instantiate a `MediaStreamSample` ❷, which is the object you'll have to return to the `MediaElement`. The `_currentVideoTimeStamp` field, containing the microseconds for the timestamp, is increased by the amount of the frame rate. You then check whether you've incremented the timestamp to a multiple of one second, and if so, flip the buffer to use when dynamically generating the frame video content ❸. You also increment the `_samplesProvided` field, which tracks the number of frames you've flipped, which is also the number of seconds the video has run.

The frame is flipped once every second, and your custom media stream source only runs for one minute. If the `_samplesProvided` field is less than 60, you pass the generated frame to the `MediaElement` via the `ReportGetSampleCompleted` ❹. If `_samplesProvided` is 60 or more, you end the stream. The stream is ended by generating an end-of-stream timestamp ❺ where the `MediaStream` object associated to the `MediaStreamSample` is null and the buffer length is zero.

You complete the `VideoMediaStreamSource` class by adding the abstract `MediaStreamSource` methods that you haven't implemented yet. You don't have any cleanup work to do, so `CloseMedia` is an empty method:

```
protected override void CloseMedia() {}
```

When you opened the media stream, you specified that your class didn't support seek operations. But `SeekAsync` will still be called with a seek value of zero if the user has pressed Stop and then Play:

```
protected override void SeekAsync(long seekToTime)
{
    if (seekToTime != 0) throw new NotSupportedException();
    _currentBufferFrame = 0;
    _currentReadyFrame = 1;
    _currentVideoTimeStamp = 0;
    _samplesProvided = 0;
    ReportSeekCompleted(seekToTime);
}
```

When `SeekAsync` is called, you validate that `seekToTime` is indeed zero. You then reset the class fields that track the current state to their initial values.

The remaining two methods aren't supported in your custom `MediaStreamSource` class. You'll implement these two methods by throwing a `NotImplementedException`:

```
protected override void SwitchMediaStreamAsync(
    MediaStreamDescription mediaStreamDescription)
{
    throw new NotImplementedException();
}
protected override void GetDiagnosticAsync(
    MediaStreamSourceDiagnosticKind diagnosticKind)
{
    throw new NotImplementedException();
}
```

You now have `VideoMediaStreamSource` fully implemented, but haven't hooked up the custom video menu option. Remember that you added a menu item to the `ApplicationBar` to load custom video, but you haven't yet implemented the menu item's click handler:

```
private void CustomVideoClicked(object sender, EventArgs e)
{
    MediaStreamSource source = new VideoMediaStreamSource(
        (int)mediaElement.ActualWidth,
        (int)mediaElement.ActualHeight);
    mediaElement.SetSource(source);
    sourceTextBlock.Text = "custom video";
}
```

The click handler implementation is fairly straightforward. You create a new `VideoMediaStreamSource` class, and pass the `MediaElement`'s height and width. You then call the `SetSource` method to load the `MediaStreamSource` instance. Finally, you update the `sourceTextBlock`.

Now that you've seen how video streams work, you're going build a custom audio stream.

12.4 Creating custom audio

You'll now build an app that dynamically generates audio samples using a custom `MediaStreamSource`. The `MediaStreamSource` interface is agnostic of the content type it's implementing, so what you've learned working with video streams also applies to audio streams. Your custom audio stream source will be a simple tuning fork simulator, which will do nothing other than generate a tone at a frequency of 440Hz. As with most of the sample code in the book, this example is designed to be as easy as possible, and doesn't pretend to be the best way to code this functionality.

Different waveforms can be used to generate a tone, and in this example you'll use a sine waveform to get a good result. You'll generate your audio samples using the Waveform Audio File Format (WAV). You use the WAV format because it's easy to use and is supported by Windows Phone.

> **NOTE** Usage of the WAV format requires you to work with an encoded `WaveFormatEx` structure. The technical details of `WaveFormatEx` are beyond the scope of this book. You can read more about `WaveFormatEx` on MSDN at http://mng.bz/FNuW.

In this example, you'll be using a class named `WaveFormatEx` which handles creating the encoded string that's specified in the `MediaStreamDescription`. We won't cover most of the details of the `WaveFormatEx` class, but you can get the class as part of the source code download.

12.4.1 Defining a custom audio stream source

You begin your custom audio source implementation by adding a new class to your MediaPlayback sample project. Name the new class `AudioMediaStreamSource` and have it inherit from `MediaStreamSource`. Add the fields shown in the next listing to the class.

Listing 12.12 Fields necessary to implement `AudioMediaStreamSource`

```
public class AudioMediaStreamSource : MediaStreamSource
{
    private MediaStreamDescription _audioDesc;
    private long _currentTimeStamp;

    private readonly int _numSamples;
    private readonly int _bufferByteCount;              ❶ Wave file
    private readonly long _audioDuration;                  data
    private readonly string _encodedWaveFormat;

    private readonly Dictionary<MediaSampleAttributeKeys, string>
        _sampleAttr = new Dictionary<MediaSampleAttributeKeys, string>();

    private readonly SineWaveformOscillator _oscillator;   <-❷ Sound generator
}
```

When working with media streams, you a need a `MediaStreamDescription` instance field, which you name `_audioDesc`. You also need to track the current timestamp, and so you create a `_currentTimeStamp` field. Next you define several read-only fields that you use to generate the media stream and audio samples ❶. For convenience, you define the media sample attribute dictionary, even though you'll never add any attributes to the dictionary. You must pass a valid dictionary when calling `ReportGetSampleCompleted`, so here you're creating an empty collection just once. The final instance field that you declare is for the `SineWaveformOscillator` ❷, which is a lightweight class used to generate sine waves.

The read-only fields are initialized in the class constructor. The `AudioMediaStreamSource` constructor, shown in the following listing, accepts a frequency parameter, allowing the calling code to determine the exact tune to play.

Listing 12.13 The `AudioMediaStreamSource` constructor

```
public AudioMediaStreamSource(short frequency)
{
    WaveFormatEx waveFormat = new WaveFormatEx()
    {
        SamplesPerSec = 44100,
        Channels = 1,
        BitsPerSample = 16,
        AvgBytesPerSec = 44100 * 2,
        BlockAlign = 2,
        FormatTag = WaveFormatEx.FormatPCM,
        Size = 0
    };
    waveFormat.ValidateWaveFormat();

    _numSamples = waveFormat.Channels * 256;
    _bufferByteCount = waveFormat.BitsPerSample / 8 * _numSamples;
    _audioDuration = waveFormat
        .AudioDurationFromBufferSize((uint)_bufferByteCount);
    _encodedWaveFormat = waveFormat.ToHexString();

    _oscillator = new SineWaveformOscillator()
    {
        Frequency = frequency * 2
    };
}
```

❶ **WaveFormatEx utility**

❷ **Initialize read-only fields**

❸ **Create oscillator**

A `WaveFormatEx` object is instantiated and initialized with the information about the audio samples ❶, and the final result is validated using the `ValidateWaveFormat` method. Next, you use the `WaveFormatEx` object to initialize the read-only fields ❷ that are used in `OpenMediaAsync` and `GetSampleAsync`. Finally, construct a `SineWaveformOscillator` object and set its `Frequency` property ❸.

The numbers used to set the `WaveFormatEx` properties represent

- 44100 Hz sampling rate
- 16-bit sample resolution
- Mono (single channel)

With the class properly constructed, let's look at how you open the audio stream.

12.4.2 Opening the audio stream source

`OpenMediaAsync` is the next method you implement for `AudioMediaStreamSource`. Just like with `VideoMediaStreamSource`, `OpenMediaAsync` must build a dictionary of `MediaSourceAttributesKeys`, and a collection of `MediaStreamDescription` objects. The next listing demonstrates how these collections are created for WAV audio stream.

Listing 12.14 `OpenMediaAsync` implementation

```
protected override void OpenMediaAsync()
{
    Dictionary<MediaStreamAttributeKeys, string> streamAttr =
        new Dictionary<MediaStreamAttributeKeys, string>();
    streamAttr[MediaStreamAttributeKeys.CodecPrivateData] =
        _encodedWaveFormat.ToHexString();

    audioDesc = new MediaStreamDescription(
        MediaStreamType.Audio, streamAttr);
```

❶ **Specify stream attributes**

```
       List<MediaStreamDescription> availableStreams =
        new List<MediaStreamDescription>();
    availableStreams.Add(_audioDesc);

    Dictionary<MediaSourceAttributesKeys, string> sourceAttr =
        new Dictionary<MediaSourceAttributesKeys, string>();
    sourceAttr[MediaSourceAttributesKeys.Duration] = "0";
    sourceAttr[MediaSourceAttributesKeys.CanSeek] = "False";

    ReportOpenMediaCompleted(sourceAttr, availableStreams);
}
```

The most significant difference between this method and `VideoMediaStreamSource` is which stream attributes you create, and the fact that you specify a `MediaStreamType` of `Audio` when you build the `MediaStreamDescription`. Instead of adding height and width stream attributes, you add an `MediaStreamAttributeKeys.CodecPrivateData` ❶ attribute with the encoded `WaveFormatEx` value you saved in the constructor.

Once the `MediaStreamSource` is opened, you're ready to provide media samples.

12.4.3 Generating audio samples

When `MediaElement` asks for media samples, you need to provide audio samples generated as a waveform. For this you use the `SineWaveformOscillator` class, as shown in the following listing.

Listing 12.15 Generating the audio sample

```
protected override void GetSampleAsync(MediaStreamType mediaStreamType)
{
    if (mediaStreamType != MediaStreamType.Audio)
        throw new NotSupportedException();

    MemoryStream stream = new MemoryStream();
    for (int i = 0; i < _numSamples; i++)
    {
        short sample = _oscillator.GetNextSample();
        stream.WriteByte((byte)(sample & 0xff));
        stream.WriteByte((byte)(sample >> 8));
    }

    MediaStreamSample streamSample = new MediaStreamSample(_audioDesc,
        stream, 0, _bufferByteCount,_currentTimeStamp, _sampleAttr);

    _currentTimeStamp += _audioDuration;

    ReportGetSampleCompleted(streamSample);
}
```

❶ **Generate sound**

Create sample ❷

❸ **Increment timestamp**

Before performing any work, you verify that the requested `MediaStreamType` is `Audio`. Next you have the core of the tone generation ❶. You ask for sound bytes from the oscillator and write them to the `MemoryStream`. You build a new `MediaStreamSample` ❷ with the populated stream, timestamp, media description, and empty sample attribute collection. Finally, the current timestamp is incremented ❸ before calling `ReportGetSampleCompleted`.

We've mentioned the `SineWaveformOscillator` class twice now, but haven't talked about how the class works. The next listing contains the entire class definition.

Listing 12.16 The `SineWaveformOscillator` class

```
public class SineWaveformOscillator
{
    double frequency;
    uint phaseAngleIncrement;
    uint phaseAngle = 0;

    public double Frequency
    {
        set
        {
            frequency = value;
            phaseAngleIncrement =
                (uint)(frequency * uint.MaxValue / 44100);
        }
        get
        {
            return frequency;
        }
    }

    public short GetNextSample()
    {
        ushort wholePhaseAngle = (ushort)(phaseAngle >> 16);
        short amplitude = (short)(short.MaxValue *
            Math.Sin(2 * Math.PI * wholePhaseAngle / ushort.MaxValue));
        phaseAngle += phaseAngleIncrement;
        return amplitude;
    }
}
```

① **Determine angle increment**

② **Calculate wave amplitude**

The two important operations are where the increment in the phase angle is determined ① and where the audio sample amplitude is calculated ②.

Coming back to the `AudioMediaStreamSource` class, you still need to implement the remaining abstract methods. Just like with `VideoMediaStreamSource`, `CloseMedia` is an empty method:

```
protected override void CloseMedia() {}
```

When `SeekAsync` is called with a zero position, you need to reset the `_currentTimeStamp`:

```
protected override void SeekAsync(long seekToTime)
{
    if (seekToTime != 0) throw new NotSupportedException();
    _currentTimeStamp = 0;
    ReportSeekCompleted(seekToTime);
}
```

The remaining two methods aren't supported in your `AudioMediaStreamSource` class. You'll implement these two methods by throwing a `NotImplementedException`:

```
protected override void SwitchMediaStreamAsync(
    MediaStreamDescription mediaStreamDescription)
{
    throw new NotImplementedException();
}

protected override void GetDiagnosticAsync(
    MediaStreamSourceDiagnosticKind diagnosticKind)
{
    throw new NotImplementedException();
}
```

You now have Audio`MediaStreamSource` fully implemented, but haven't hooked up the custom audio menu option. Remember that you added a menu item to the `ApplicationBar` to load custom audio, but haven't yet implemented the menu item's click handler:

```
private void CustomAudioClicked(object sender, EventArgs e)
{
    MediaStreamSource source = new AudioMediaStreamSource(440);
    mediaElement.SetSource(source);
    sourceTextBlock.Text = "custom audio";
}
```

In the `CustomAudioClicked`, when you create the audio stream source, you pass in a frequency of 440 Hz, which is the standard tone for the musical note middle A.

Up to this point we've explored the `MediaElement` control and its media download and playback capabilities. We're now going to take a look at another Microsoft technology that can be used to deliver streaming media to the Windows Phone.

12.5 Streaming media clients

Microsoft's Smooth Streaming technology mixes IIS 7 extensions, Silverlight client libraries, and a robust API to enable streaming media experiences such as live viewing of sporting events or conferences, as well as movie delivery. How does streaming media differ from the more traditional media download and playback?

Media players like the `MediaElement` control rely on local copies of media files. When a remote file is specified, the media player downloads the file and begins playback once the entire file is available. To avoid waiting for the entire file to be available, most media players enhance the playback experience by using *progressive download* techniques. Progressive download works by playing the media as the file is downloaded. If the download speed is fast enough, the multimedia file will be rendered with no problem. Because progressive download is file-based, the quality and resolution of the media must be specified when the media is opened, and isn't adjusted when network quality changes during playback. What happens when you start watching a video on your phone, then go out of range of a wireless network?

Streaming media solutions do account for network quality. The media served up by a streaming media server exists in several resolutions and quality formats. A streaming media client will determine the capabilities of the host device, and will monitor network quality. The client will then ask the server for a media stream that's appropriate for current conditions. Where streaming media differs from progressive download is that the client can adapt the quality of the media when network conditions change. When your phone leaves a wireless network, a streaming media client will adjust to the change and will ask for a lower-quality video stream.

Streaming media standards

Streaming media standards have been created, and their adoption usually requires the usage of three different protocols to handle session management, multimedia data transport, and quality of service.

IETF RFCs define the RTSP protocol for session management (RFC 2326), a simple plain-text protocol which provides VCR-like commands to a server about the multimedia session in progress. The RTP protocol (RFC 1889) defines how multimedia data should be transported. The RTCP protocol (RFC 1889) defines the control of the RTP traffic.

Unfortunately there's no native support for these protocols in Silverlight. In theory you could use a custom `MediaStreamSource` to stream media to the `MediaElement` control, generating the raw TCP/UDP packets required by the RTSP/RTP/RTCP protocols.

Microsoft has implemented a streaming technology named *Smooth Streaming* that communicates over the HTTP protocol and is compatible with Silverlight and the limitations of the Windows Phone. On the server side, Smooth Streaming is an extension to Internet Information Server 7 (IIS). Smooth Streaming works with contiguous files stored in a fragmented MPEG-4 (MP4) media container format, which are created by encoders that use a Smooth Streaming profile to encode the video source. A Smooth Streaming server provides different quality levels for the same media streams.

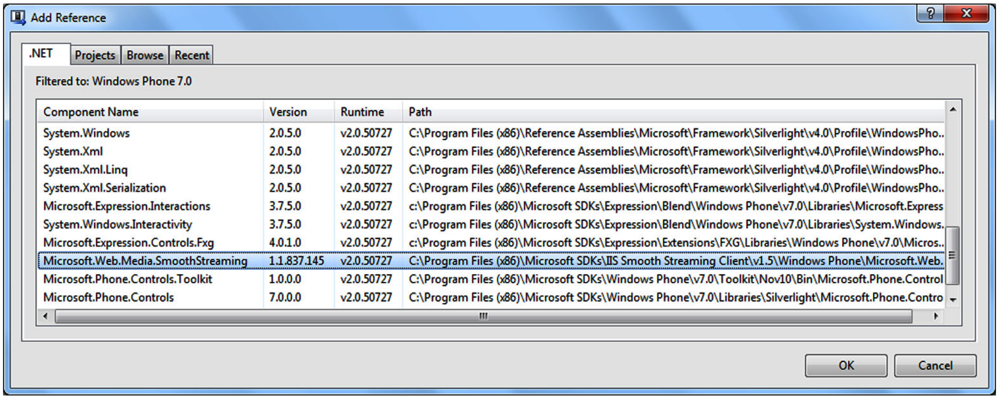
NOTE The implementation of a Smooth Streaming server as well as building streaming media are beyond the scope of this book.

Microsoft's Smooth Streaming technology has already been used in several commercial applications including the live broadcasting of the Olympic Games in 2008 and 2010. For a detailed description of the Smooth Streaming technology, you can visit http://mng.bz/1a2Y.

Smooth Streaming can be broken out into three categories—IIS extensions, Silverlight clients, and media encoding. In this book we focus solely on the Silverlight client libraries. You can download the IIS Smooth Streaming Client 1.5 libraries from http://mng.bz/N2ny. You'll use the client libraries to build the next sample application.

12.5.1 Using Smooth Streaming

Silverlight applications for Windows Phone support Smooth Streaming technology by using the `SmoothStreamingMediaElement` control. You'll build a new sample application using the Windows Phone Application project template and name the project SmoothStreaming. The next step is to add a reference to the Smooth Stream Client



Filtered to: Windows Phone 7.0

Component Name	Version	Runtime	Path
System.Windows	2.0.5.0	v2.0.50727	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\Silverlight\v4.0\Profile\WindowsPho..
System.Xml	2.0.5.0	v2.0.50727	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\Silverlight\v4.0\Profile\WindowsPho..
System.Xml.Linq	2.0.5.0	v2.0.50727	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\Silverlight\v4.0\Profile\WindowsPho..
System.Xml.Serialization	2.0.5.0	v2.0.50727	C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\Silverlight\v4.0\Profile\WindowsPho..
Microsoft.Expression.Interactions	3.7.5.0	v2.0.50727	c:\Program Files (x86)\Microsoft SDKs\Expression\Blend\Windows Phone\v7.0\Libraries\Microsoft.Express
System.Windows.Interactivity	3.7.5.0	v2.0.50727	c:\Program Files (x86)\Microsoft SDKs\Expression\Blend\Windows Phone\v7.0\Libraries\System.Windows.
Microsoft.Expression.Controls.Fxg	4.0.1.0	v2.0.50727	c:\Program Files (x86)\Microsoft SDKs\Expression\Extensions\FXG\Libraries\Windows Phone\v7.0\Micros..
Microsoft.Web.Media.SmoothStreaming	1.1.837.145	v2.0.50727	C:\Program Files (x86)\Microsoft SDKs\IIS Smooth Streaming Client\v1.5\Windows Phone\Microsoft.Web.
Microsoft.Phone.Controls.Toolkit	1.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.0\Toolkit\Nov10\Bin\Microsoft.Phone.Control
Microsoft.Phone.Controls	7.0.0.0	v2.0.50727	C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v7.0\Libraries\Silverlight\Microsoft.Phone.Contro

Figure 12.3 Adding a reference to the Smooth Streaming Client assembly

assembly (see figure 12.3) which is named Microsoft.Web.Media.SmoothStreaming.dll. The assembly should be installed under your Microsoft SDK folder at the path C:\Program Files\Microsoft SDKs\IIS Smooth Streaming Client\v1.5\Windows Phone. The Microsoft SDK folder is located under Programs Files (x86) if you have a 64-bit operating system.

The SmoothStreaming sample application will function similarly to the MediaPlayback application you built earlier in this chapter. You'll reuse much of the XAML markup and code behind that you created for MediaPlayback. Open MediaPlayback's MainPage.xaml and copy the entire contents of the `ContentPanel`, and paste the markup into SmoothStreaming's MainPage.xaml. You don't need the `ApplicationBar` menu items, so delete them, but make sure to leave the buttons.

You want to do the same thing for the code behind. Open the version of MainPage.xaml.cs in MediaPlayback and copy all of the event handlers, except for the menu item clicked handlers. Paste the copied code into SmoothStreaming's MainPage.xaml.cs. After you've copied this code, go ahead and close the MediaPlayback project files.

Now let's return to SmoothStreaming's MainPage.xaml file. You're going to change the existing `MediaElement` control into a `SmoothStreamingMediaElement` control. You need a proper namespace added to the page where you'll be using this control in order to have it available in the XAML:

```
xmlns:smoothStreaming="clr-namaespace:Microsoft.Web.Media.SmoothStreaming;
➥ assembly=Microsoft.Web.Media.SmoothStreaming"
```

The namespace should be included after the declaration of the other namespaces in the MainPage.xaml file. Now you can replace the `MediaElement` control:

```
<smoothStreaming:SmoothStreamingMediaElement x:Name="mediaElement"
    SmoothStreamingSource="http://mediadl.microsoft.com/mediadl/iisnet/
➥ smoothmedia/Experience/BigBuckBunny_720p.ism/Manifest" ... />
```

In order to set the media source, you use a different property than for the progressive download scenario you used with the `MediaElement` control. In this case you set the `SmoothStreamingSource` property. The `SmoothStreamingSource` property is a `Uri` that references a Smooth Streaming manifest file. The manifest file describes the various resolutions and bitrates that are available for streaming.

The `SmoothStreamingMediaElement` is similar to `MediaElement`—similar enough that you don't have to change the implementation of the `PlayClicked`, `PauseClicked`, `StopClicked`, `MuteClicked`, or `MediaFailed` event handlers. The data binding between the media element, the volume `Slider`, and the position `TextBlock` also remain unchanged.

The only event handler you need to change is for the `CurrentStateChanged` event. The new implementation of the event handler is shown in the following listing.

Listing 12.17 The new `CurrentStateChanged` event handler

```
private void mediaElement_CurrentStateChanged(
    object sender, RoutedEventArgs e)
{
    stateTextBlock.Text = mediaElement.CurrentState.ToString();
    TrackInfo track = mediaElement.VideoPlaybackTrack;

    if (track != null)
    {
        string widthAttr;
        string heightAttr;
        track.Attributes.TryGetValue("Width", out widthAttr);
        track.Attributes.TryGetValue("Height", out heightAttr);
        sourceTextBlock.Text = string.Format("{0}x{1} {2}",
            widthAttr, heightAttr, track.Bitrate);
    }

    if (mediaElement.CurrentState ==
        SmoothStreamingMediaElementState.Opening)
        mediaProgress.Visibility = Visibility.Visible;
    else
        mediaProgress.Visibility = Visibility.Collapsed;
}
```

❶ **Get video track**

❷ **Look up resolution**

❸ **Use SmoothStreamingMediaState**

A Smooth Streaming source provides a variety of different tracks, each with their own characteristics. The media element will choose the track that best matches the network quality. You're going to reuse the `sourceTextBlock` to display the resolution and bitrate of the currently playing track. You get a reference to the current track via the `VideoPlaybackTrack` property ❶, which returns a `TrackInfo` object. Like other media classes, `TrackInfo` contains a dictionary of attribute values. You read the width and height attributes from the current track's dictionary ❷, and then update the `sourceTextBlock`.

`SmoothStreamingMediaElement`'s `CurrentState` property is of type `SmoothStreamingMediaElementState`, which requires you to make a minor change to the Boolean expression ❸, as the type is different from that used in `MediaElement`'s `CurrentState` property.

The Smooth Streaming media that has been created before now has all been designed for Silverlight for the browser, containing tracks that match up with the codecs, resolutions, and bitrates supported by the browser. Depending on how the media was designed and encoded, it may not run on the Windows Phone due to limitations in the platform.

12.5.2 Streaming limitations

Windows Phone doesn't support all of the resolutions, bitrates, and video formats that are supported by Silverlight for the browser. The maximum resolution supported by the phone is 720 wide by 480 high. Audio streams have their own limits, and the upper bounds for audio and video fluctuate with frames per second and other attributes, so consult the IIS Smooth Streaming documentation for specifics.

With many streaming media scenarios, different quality videos are encoded at different resolutions and bitrates. When the network quality changes during playback, the media player will begin playing a file of a different resolution. The phone can't support variable resolutions during playback, and different quality streams must share the same resolution before the phone can use them in adaptive streaming.

These limitations require streaming solutions to either build phone-specific media streams on the server, or add extra handling on the client. For the client-side implementation, code must be written to examine the Smooth Streaming manifest file and restrict the available tracks to only those supported by the phone. The manifest information should be examined once the manifest file has been downloaded by the `SmoothStreamingMediaElement` control. When the manifest file has been downloaded and processed the control raises the `ManifestReady` event. Update the MainPage.xaml and subscribe to the event:

```
<smoothStreaming:SmoothStreamingMediaElement x:Name="mediaElement" ...
    ManifestReady="mediaElement_ManifestReady" />
```

In your `ManifestReady` event handler, shown in the next listing, you're going to loop through the tracks listed in the manifest and find the tracks with the highest supported resolution. You're then going to tell the `SmoothStreamingMediaElement` to only use those tracks for playback.

Listing 12.18 Processing the smooth streaming manifest

```
private void mediaElement_ManifestReady(object sender, EventArgs e)
{
    const ulong maxSupportedHeight = 480;
    const ulong maxSupportedWidth = 720;

    foreach (SegmentInfo segment in mediaElement.ManifestInfo.Segments)
    {
        foreach (StreamInfo streamInfo in segment.AvailableStreams)
        {
            if (streamInfo.Type == MediaStreamType.Video)
            {
```

① **Maximum resolution supported by phone**

```
                List<TrackInfo> usableTracks = new List<TrackInfo>();
                ulong bestHeight = 0;
                ulong bestWidth = 0;

                foreach (TrackInfo track in streamInfo.AvailableTracks)
                {
                    string widthAttr;
                    string heightAttr;
                    ulong width = 0;
                    ulong height = 0;

                    if (track.Attributes.TryGetValue("Width",
                        out widthAttr))
                    {
                        ulong.TryParse(widthAttr, out width);
                    }

                    if (track.Attributes.TryGetValue("Height",
                        out heightAttr))
                    {
                        ulong.TryParse(heightAttr, out height);
                    }

                    if (height > 0 && width > 0 &&
                        height <= maxSupportedHeight &&
                        width <= maxSupportedWidth)
                    {
                        if (height > bestHeight || width > bestWidth)
                        {
                            bestHeight = height;
                            bestWidth = width;
                            usableTracks.Clear();
                            usableTracks.Add(track);
                        }
                        else if(height == bestHeight && width == bestWidth)
                        {
                            usableTracks.Add(track);
                        }
                    }
                }
                streamInfo.RestrictTracks(usableTracks);
            }
        }
    }
}
```

② **Look up resolution**

③ **Check resolution**

④ **Found better resolution**

⑤ **Report supported tracks**

The maximum resolution you're going to accept is 720 x 480 ①. After download, the manifest is processed by the media element control and exposed by the `ManifestInfo` property. The manifest is broken into segments and each segment contains a collection of streams. You check each stream and ignore those that aren't video streams. Each stream has a collection of tracks that's accessible via the `AvailableTracks` property of the `StreamInfo` class. You loop through each of the tracks looking for the highest-resolution track that meets your maximum resolution limitation. You read the width and height attributes from the current track's dictionary ②. You then check the width and height to see that they're greater than zero and less than the maximum supported resolution ❸. If the track's resolution is greater than your current best resolution, clear out the `usableTracks` list and remember the current track and resolution ❹. If the track's resolution is the same as the current best resolution, add the track to the `usableTracks` list. Once you've processed all the tracks in the stream, you call the `RestrictTracks` method ❺ to request that the media element control only use the tracks you've picked.

The algorithm presented in `mediaElement_ManifestReady` only restricts tracks based on the resolution of the track. Other attributes of the stream might be considered when choosing tracks. Smooth Streaming media may use codecs or bitrates that aren't supported by the phone. You may also have nontechnical factors to consider when limiting media content. For example, you may sell subscriptions to your content, and may choose to limit nonpaying users to lower-quality media.

We've covered some of the basic topics with the Smooth Streaming Client that you need to be aware of when building applications for Windows Phone. There are many more Smooth Streaming topics we haven't discussed, and you should consult the Smooth Streaming documentation.

12.6 Summary

`MediaElement` and `SmoothStreamingMediaElement` allow Windows Phone developers to create cool multimedia applications incorporating audio and video, whether building a full media player or just incorporating clips into a Silverlight game or application.

Windows Phone supports a wide variety of audio and video containers and decoders, but does have its limitations. Knowing the limits of the phone platform and of each of the media elements is important when working with your application teams. You don't want to find out that the media clip your animators and designers just built is in a format that won't work on the phone.

In this chapter we looked at `MediaElement`, one of Silverlight's advanced controls. In the next chapter, we take a deep dive into two other advanced Silverlight controls—the Bing Maps Control and the `WebBrowser`.

13

Using Bing Maps and the browser

This chapter covers

- Using the Bing Maps control and web services
- Determining device location
- Using the `WebBrowser` control
- Building HTML 5 applications

Windows Phone includes two applications that are essential for mobile platforms—Bing Maps and Internet Explorer 9. The mobile edition of Internet Explorer 9 supports HTML 5 and hardware-accelerated graphics. Bing Maps provides the user with maps, directions, and searches for nearby businesses and landmarks. The Windows Phone SDK enables developers to access these technologies from within their applications.

Both Bing Maps and Internet Explorer can be started from within an application using tasks that launch the native applications, pushing the running application to the background. When a more integrated user experience is desired, the developer can embed a map or browser into an application using Silverlight controls provided in the SDK.

In this chapter we'll demonstrate how to use the location service to pinpoint a device's actual position on the globe inside of a location-aware application. In addition to providing latitude and longitude, the location service can also provide altitude, speed, and heading. The location service is more than a simple GPS sensor. It combines GPS, Wi-Fi, and cellular network data with a web service to provide location information. You'll combine the location service with the Silverlight `Map` control and explore how to display an embedded map pinpointing the user's location and tracking their movements. We'll also go into detail about how to leverage the Bing Maps SOAP web services to determine the user's physical address using their current location.

Version 7.1 of the Windows Phone SDK added two new Bing Maps launchers to provide developers with an easy way to integrate maps into an application. The new launcher tasks are `BingMapsTask`, a launcher for the Bing Maps application, and `BingMapsDirectionsTask`, which launches the Bing Maps application and displays driving directions between two points.

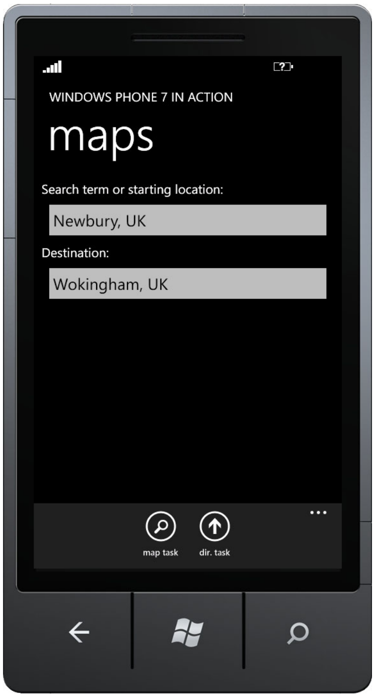
Internet Explorer can also be integrated with an application using a launcher task. The `WebBrowserTask` allows the developer to start Internet Explorer and to specify the `Uri` of a web page to be loaded into the browser. Internet Explorer can be embedded into an application using the Silverlight `WebBrowser` control. We'll discuss both the rendering of HTML 5 within an application and how C# code in the application can interoperate with JavaScript in the web page.

To highlight Internet Explorer and Bing Maps, you're going to build three different sample applications. In addition to building the location sample application, we'll demonstrate how to use the `WebBrowser` control to build an HTML 5 and JavaScript-based application. First we'll introduce Bing Maps by showing you how to build a sample that launches the two Bing Maps tasks.

13.1 Introducing Bing Maps

Applications can include mapping features by either launching the native Bing Maps application or by embedding the Silverlight `Map` control into the application user interface. In this section you'll build a simple application to demonstrate how to use the `BingMapsTask` and `BingMapsDirectionsTask`. The sample application, shown in figure 13.1, prompts the user to enter a search term or two locations and allows the user to launch the native Bing Maps application to show the specific location and driving directions.

You'll get started by preparing a new project with the basic user interface controls and buttons required to allow the user to interact with Bing Maps.


WINDOWS PHONE 7 IN ACTION

maps

Search term or starting location: Newbury, UK

Destination: Wokingham, UK

Figure 13.1 The MapsTasks sample application

13.1.1 Preparing the application

Your first sample application is named MapsTasks, and is based off of the Windows Phone Application project template. Open Visual Studio and create the new project. The main user interface prompts the user to enter a search term or starting location and a destination location. These locations aren't necessarily full addresses, and are passed to Bing Maps as search terms. Open MainPage.xaml and add two `TextBlocks` and two `TextBoxes` to the `ContentPanel`:

```
<StackPanel>
    <TextBlock Text="Search term or starting location:" />
    <TextBox x:Name="departureTerm" />
    <TextBlock Text="Destination:" />
    <TextBox x:Name="destinationTerm" />
</StackPanel>
```

The `TextBoxes` should be named `departureTerm` and `destinationTerm` to allow you to access their `Text` properties from code behind in MainPage.xaml.cs.

You also need two buttons to enable two different features in the application. The first feature opens the native Bing Maps application with the starting location centered in the map. The second feature opens the native Bing Maps application showing driving directions between the starting and destination locations:

```
<shell:ApplicationBarIconButton Text="map task" Click="mapTask_Click"
    IconUri="/Images/appbar.feature.search.rest.png" />
<shell:ApplicationBarIconButton Text="dir. task" Click="dirTask_Click"
    IconUri="/Images/appbar.directions.png" />
```

Go ahead and create empty click event handlers for both of the buttons. You'll add implementations for each click event handler as you progress through the section. The first click event handler will launch the native Bing Maps application using the `BingMapsTask`.

13.1.2 Launching the Bing Maps application

The native Bing Maps application can be launched from any third-party application using the `BingMapsTask` launcher class. You learned how to use launchers and choosers in chapter 4. The `BingMapsTask` launcher exposes three properties to determine its behavior—`Center`, `SearchTerm`, and `ZoomLevel`.

Use the `Center` property to determine where the map is to be centered. If no value is specified for the `Center` property, Bing Maps will attempt to center the map at the device's current location. The actual centering behavior is influenced by the search term and zoom level values specified when the task is launched, as well as locations used in previous searches.

The optional `SearchTerm` property is a string used to highlight specific points on the map. The search term might contain a full or partial address, a city name, or the name of a landmark. The `SearchTerm` might specify other type of searches as well. For example, the user could type in *Pizzeria* in order to find the closest restaurant serving pizzas. Locations matching the search term are identified on the map with a push pin, as shown in figure 13.2.

The `ZoomLevel` controls the initial zoom level to be used to display the map. The `BingMapsTask` documentation doesn't clarify what the appropriate values are for a zoom level, but our experimentation suggests that reasonable values are between 10 and 20.

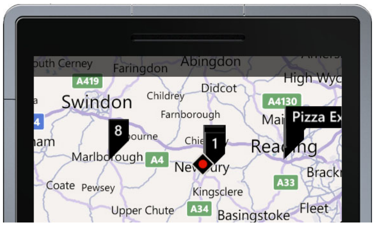


Figure 13.2 Searching for a local pizzeria

The MapsTasks sample application uses the `BingMapsTask` when the user taps the first button in the application bar. The following listing shows the implementation of the button's `Click` event handler.

Listing 13.1 Launching Bing Maps

```
private void mapTask_Click(object sender, EventArgs e)
{
    if(string.IsNullOrEmpty(departureTerm.Text))
    {
        MessageBox.Show("Please enter a start location.");
        return;
    }

    var task = new BingMapsTask
    {
        SearchTerm = departureTerm.Text,
        ZoomLevel = 15,
    };
    task.Show();
}
```

① Check for valid search term

② Launch Bing Maps

Before you do any work, you check whether the user has entered a valid search term ①. The `BingMapsTask` requires either the `Center` property or the `SearchTerm` property to be set, and will throw an `InvalidOperationException` if both of them are empty. If the search term is valid, you construct a new instance of the `BingMapsTask` and set the `SearchTerm` property to the value of the `departureTerm.Text` field. You also hard-code the `ZoomLevel` to 15. Finally you launch the Bing Maps application with a call to the `Show` method ②.

The application is now ready to test your first feature. Run the application on either your device or the emulator, enter a search term, and tap the map task button.

Now let's move on to implementing the second feature: displaying directions between two locations.

13.1.3 *Finding directions*

`BingMapsTask` doesn't come alone, and a second Bing Maps launcher named `BingMapsDirectionsTask` provides an easy way to get directions from Bing Maps. `BingMapsDirectionsTask` can be customized using two properties named `Start` and

`End`. Both these properties are of the type `LabeledMapLocation` and at least one of the two must be set or `BingMapsDirectionsTask` will throw an `InvalidOperationException` when launched. When the `BingMapsDirectionsTask` is launched and either the `Start` or `End` property isn't set, Bing Maps will use the device's current position in place of the missing property.

The `LabeledMapLocation` class is used to provide a geographic coordinate along with a label for the location. If a `GeoCoordinate` isn't specified in the `LabledMapLocation`'s `Location` property, `BingMapsDirectionsTask` interprets the `Label` property in much the same way that `BingMapsTask` interprets its `SearchTerm` property.

The MapsTasks sample application uses the `BingMapsDirectionsTask` when the user taps the second button in the application bar. The button's `Click` event handler is shown in the next listing.

Listing 13.2 Launching Bing Maps to calculate directions

```
private void directionTask_Click(object sender, EventArgs e)
{
    LabeledMapLocation start = null;
    LabeledMapLocation end = null;

    if (!string.IsNullOrEmpty(departureTerm.Text))
        start = new LabeledMapLocation { Label = departureTerm.Text };

    if (!string.IsNullOrEmpty(destinationTerm.Text))
        end = new LabeledMapLocation { Label = destinationTerm.Text };

    if (start == null && end == null)
    {
        MessageBox.Show("Please enter start and/or end locations.");
        return;
    }

    var task = new BingMapsDirectionsTask { Start = start, End = end };
    task.Show();
}
```

Did user input search term? ①

② **Check for valid locations**

③ **Launch Bing Maps**

You start the listing by declaring two `LabeledMapLocation` variables, one for the departure location and the other for the destination location. You only construct `LabeledMapLocations` when the user has entered a term in the related `TextBox` controls ①. To avoid an `InvalidOperationException`, you check to see that at least one valid location was created ②. Finally you construct the `BingMapsDirectionsTask` and launch the native Bing Maps application ③.

Test your second feature by running the application, entering two search terms, and tapping the directions task button. Press the Back button to return to the application, delete one of the terms, and tap the directions task button again. Continue to return to the application to enter different search terms until you're comfortable with how the `BingMapsDirectionsTask` works with various combinations of search terms.

The Bing Maps tasks are ideal for adding simple mapping features into your application. The downside of using the Bing Maps tasks is that your application becomes dormant when the Bing Maps application is launched. The next sample application shows you how to combine the Silverlight `Map` control with location services to provide a rich map experience inside an application.

13.2 Location services

The location service is made up of different bits of hardware, software, and web services. The hardware includes a built-in global positioning system (GPS) receiver, the cellular radio, and the wireless network adapter. The web service is Microsoft Location Service, which fronts a database that records the coordinates of wireless access points. The data from the web service, GPS, and cellular radio is analyzed to calculate the phone's current longitude, latitude, and altitude.

All of this complexity is hidden behind the interface of a single class named `GeoCoordinateWatcher`. The `GeoCoordinateWatcher` class is found in the `System.Device.Location` namespace, which is located in the System.Device.dll assembly. `GeoCoordinateWatcher` provides the properties, methods, and events an application uses to read location data. The service is started with a `Start` method, and stopped with a `Stop` method. The current location is read from the `Position` property and the class raises the `PositionChanged` event whenever the position changes.

> **NOTE** The `ID_CAP_LOCATION` capability must be declared in the WMAppManifest.xml file in order to use the `GeoCoordinateWatcher`.

The `Position` property returns an instance of the `GeoCoordinate` class. The `GeoCoordinate` class exposes `Longitude`, `Latitude`, and `Altitude` properties to determine the device location. `Speed` and `Course` are also available to determine the device's relative motion.

The GPS receiver is the most accurate source of location data. This increased accuracy comes with a price. Activating the GPS receiver is an expensive operation and the GPS uses a lot of battery power. Determining location from the cellular radio or the network adapter uses less battery power, with the tradeoff of decreased accuracy. The `GeoCoordinateWatcher` allows the developer to specify the accuracy of location data with the `DesiredAccuracy` property. The `DesiredAccuracy` can be set to one of the two values declared in the `GeoPositionAccuracy` enumeration: `Default` or `High`. When reading data, the accuracy of the `GeoCoordinate` can be determined using the `IsUnknown`, `HorizontalAccuracy`, and `VerticalAccuracy` properties. Both the `HorizontalAccuracy` and `VerticalAccuracy` properties return the accuracy range in meters.

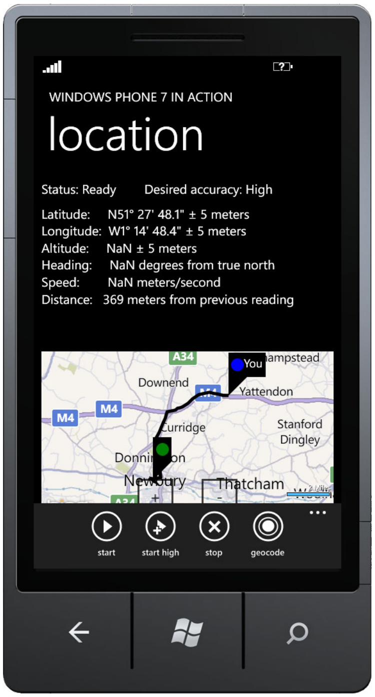
To demonstrate how to use the `GeoCoordinateWatcher` you'll create a new sample application.

13.2.1 Building the sample application

Start the new sample application by creating a new Windows Phone Application named LocationAndMaps. After the project is created, add references to the Microsoft.Phone.Controls.Maps.dll and System.Device.dll assemblies. `Microsoft.Phone.Controls.Maps` contains the Silverlight `Map` control and its related classes. `System.Device` provides the `GeoCoordinateWatcher`. The application is fairly simple, displaying two `TextBlocks` and a `Map`. A screenshot of the application is shown in figure 13.3.

The first `TextBlock` displays the `Status` reported by `GeoCoordinateWatcher`, and the second `TextBlock` displays information about the current location. The two `TextBlocks` are placed in a `StackPanel`, which is placed in the `ContentPanel` in MainPage.xaml:

```
<Grid x:Name="ContentPanel" Grid.Row="1"
     Margin="12,0,12,0">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <StackPanel>
    <TextBlock x:Name="status" />
    <TextBlock x:Name="position" />
  </StackPanel>
</Grid>
```



WINDOWS PHONE 7 IN ACTION

location

Status: Ready Desired accuracy: High

Latitude: N51° 27' 48.1" ± 5 meters
Longitude: W1° 14' 48.4" ± 5 meters
Altitude: NaN ± 5 meters
Heading: NaN degrees from true north
Speed: NaN meters/second
Distance: 369 meters from previous reading

Figure 13.3 The LocationAndMaps sample application

The sample application uses the `ApplicationBar` to present the user with four buttons. The first button starts the location service with default accuracy and the second button starts the service with high accuracy. The third button stops the location service. The last button displays the user's physical address by reverse geocoding the current location:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="False">
        <shell:ApplicationBarIconButton Click="startDefault_Click"
            Text="start" IconUri="/Images/appbar.transport.play.rest.png"/>
        <shell:ApplicationBarIconButton Click="startHigh_Click"
            Text="start high" IconUri="/Images/appbar.play.high.png" />
        <shell:ApplicationBarIconButton Click="stop_Click" Text="stop"
            IconUri="/Images/appbar.cancel.rest.png" />
        <shell:ApplicationBarIconButton Text="geocode"
            IconUri="/Images/appbar.target.png" Click="geocode_Click" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

The images for the first and third buttons come from the icons library included with the Windows Phone SDK. The second and fourth buttons use images that can be found in the book's sample source code. Create a project folder named Images and add the four images to the folder, setting their build action property to `Content`.

The four click event handlers are shown in the following listing. The event handlers each call a method that you'll implement in the next section.

Listing 13.3 Start and stop click event handlers

```
private void startDefault_Click(object sender, EventArgs e)
{
    StartService(GeoPositionAccuracy.Default);
}
private void startHigh_Click(object sender, EventArgs e)
{
    StartService(GeoPositionAccuracy.High);
}
private void stop_Click(object sender, EventArgs e)
{
    StopService();
}
private void geocode_Click(object sender, EventArgs e){}
```

① **Use default accuracy**

② **Use high accuracy**

Both of the start click event handlers call the same `StartService` method. The difference between the two is that `startDefault_Click` passes the `Default` accuracy value ① whereas the `startHigh_Click` method passes the `High` ② accuracy value.

You now have the skeleton in place for the sample application. In addition to implementing the `StartService` and `StopService` methods, you need to create a `GeoCoordinateWatcher` and hook up the service to the rest of the application.

13.2.2 Hooking up the service

You construct and initialize an instance of the `GeoCoordinateWatcher` in the `StartService` method. The accuracy of the service is specified during construction of the `GeoCoordinateWatcher` and can't be changed. Your sample application allows the user to start the service in either high accuracy or low accuracy modes, so you must create a new instance of the `GeoCoordinateWatcher` when the service is started.

The location service is started in the `StartService` method, which is shown in the next listing. The `StartService` method is called by both start button click event handlers, and passed the desired `GeoPositionAccuracy` value.

Listing 13.4 Initializing and starting the location service

```
GeoCoordinateWatcher service;
private void StartService(GeoPositionAccuracy accuracy)
{
    if (service != null)
        StopService();

    service = new GeoCoordinateWatcher(accuracy);
    service.MovementThreshold = 1.0;
    service.PositionChanged += service_PositionChanged;
    service.StatusChanged += service_StatusChanged;

    status.Text = string.Format("Permission: {0}\n", service.Permission);
    position.Text = string.Empty;
```

① **Class-level field**

② **Set threshold**

```
    if (service.Permission == GeoPositionPermission.Granted)
        service.Start();
}
```

Has permission been granted by user? ❸

You start by creating a new class-level field named service ❶ and defining the `StartService` method. After destroying any existing `GeoCoordinateWatcher` instance, you construct a new service and pass in the specified `GeoPositionAccuracy` value. Next you set the movement threshold to one meter ❷ and subscribe to the `PositionChanged` and `StatusChanged` events. Finally, you check the `Permission` property and start the service if the user has granted permission ❸ for your application to use the location service.

Starting the location service isn't an instantaneous operation and the `Start` method relegates the work to a background thread. The `GeoCoordinateWatcher` reports its status via the `Status` property or the `StatusChanged` event. The `Status` is reported as one of the values in the `GeoPositionStatus` enumeration and will be `Disabled`, `Ready`, `Initializing`, or `NoData`. The sample application subscribes to the `StatusChanged` event and displays the current status and accuracy to the user:

```
void service_StatusChanged(object sender,
    GeoPositionStatusChangedEventArgs e)
{
    status.Text = string.Format("Status: {0}  Desired accuracy: {1}",
        e.Status, service.DesiredAccuracy);
}
```

You also use the `Status` property when reporting the current `Position` to the user in the `PositionChanged` event handler. The `service_PostionChanged` method, shown in the following listing, only updates the screen when the service status is `Ready`.

Listing 13.5 Handling the `PositionChanged` event

```
GeoCoordinate previous = new GeoCoordinate();

void service_PositionChanged(object sender,
    GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    if (service.Status == GeoPositionStatus.Ready)
    {
        GeoCoordinate location = e.Position.Location;
        UpdatePositionText(location);
        previous = location;
    }
}
```

❶ **Remember previous coordinate**

❷ **Only update if sensor is ready**

❸ **Update user interface**

You start by creating a new class-level `GeoCoordinate` field named `previous` ❶ to remember the last position reported by the location service. You'll use the previous position to inform the user how much distance was covered between the current reading and the previous reading. Next you implement the `service_PostionChanged` event handler. After checking to see that the service is `Ready` ❷, you read the current `Location` from the `GeoPositionChangedEventArgs`. You update the user interface ❸ by calling the `UpdatePositionText` method. The implementation of `UpdatePositionText` is shown in the next listing.

Listing 13.6 Display location in the user interface

```
private void UpdatePositionText(GeoCoordinate loc)
{
    StringBuilder b = new StringBuilder();
    b.AppendFormat("Latitude: {0} ± {1:F0} meters\n",
        FormatCoordinate(loc.Latitude, 'N', 'S'),
        loc.HorizontalAccuracy);
    b.AppendFormat("Longitude: {0} ± {1:F0} meters\n",
        FormatCoordinate(loc.Longitude, 'E', 'W'),
        loc.HorizontalAccuracy);
    b.AppendFormat("Altitude: 0:F0} ± {1:F0} meters\n",
        loc.Altitude, loc.VerticalAccuracy);
    b.AppendFormat("Heading: {0:F0} degrees from true north\n",loc.Course);
    b.AppendFormat("Speed: {0:F0} meters/second\n", loc.Speed);

    double distance = Double.NaN;
    if (!loc.IsUnknown && !previous.IsUnknown)
        distance = loc.GetDistanceTo(previous);
    b.AppendFormat("Distance: {0:F0} meters from previous reading\n\n",
        distance);
    position.Text = b.ToString();
}
```

❶ **Format coordinate**

❷ **Calculate distance traveled**

The location properties are read from the `GeoCoordinate` and appended along with message text to a `StringBuilder` instance. When writing the longitude and latitude, you call a method named `FormatCoordinate` ❶. In the middle of creating the message text, you use the `GetDistanceTo` method ❷ to calculate the distance traveled since the previous value was read. Finally you update the user interface.

The location service reports longitude and latitude as floating-point numbers representing the number of degrees east or west of the prime meridian and the number of degrees north or south of the equator. Positive latitude values represent a position north of the equator. Positive longitude values represent a position east of the Prime Meridian. A few common formats are used to display degrees, and you've chosen the format that breaks the number into degrees, minutes, and seconds:

N37° 24' 7.9"
W122° 8' 38.1"

The `FormatCoordinate` method accepts an angle measurement in degrees, along with the characters to display for positive and negative values, and returns a string displaying direction, degrees, minutes, and seconds. The following listing shows the implementation of `FormatCoordinate`.

Listing 13.7 Formatting longitude or latitude

```
private string FormatCoordinate(double coordinate,
    char positive, char negative)
```

```
{
    char direction = coordinate >= 0 ? positive : negative;

    coordinate = Math.Abs(coordinate);
    double degrees = Math.Floor(coordinate);
    double minutes = Math.Floor((coordinate - degrees) * 60.0D);
    double seconds = (((coordinate - degrees) * 60.0D) - minutes) * 60.0D;

    string result = string.Format("{0}{1:F0}° {2:F0}' {3:F1}\"",
        direction, degrees, minutes, seconds);
    return
     result;
}
```

① **Determine character for direction**

② **Convert decimal portion into minutes**

You start by picking the character that should be shown ①. For example, the calling code should pass in N and S for latitude, and if the coordinate value is negative, S should be shown. Next you break the value into whole degrees, minutes, and seconds. You use the `Math.Floor` method to return the whole part of a number. To calculate the number of minutes ② you subtract out the whole part of the coordinate value and multiply by 60. You do a similar operation to calculate the number of minutes. Finally you format the numbers into a string.

Before you can finish the LocationService application, you need to implement the `StopService` method. The `StopService` method, shown in the next listing, stops and disposes of the `GeoCoordinateWatcher`.

Listing 13.8 Stopping the location service

```
private void StopService()
{
    if (service != null)
    {
        service.Stop();
        service.PositionChanged -= service_PositionChanged;
        service.StatusChanged -= service_StatusChanged;
        service.Dispose();
        service = null;
        status.Text += "\nThe Location Service has been stopped.";
    }
}
```

① **Unhook events**

② **Update user interface**

Before releasing the service, you call the `Stop` method and unhook the event handlers ①. Next you call the `Dispose` method and set the `service` to `null`. Before exiting the method, you update the user interface ② to inform the user that the service has actually been stopped.

Run the application and start the location service. Experiment by walking several yards in both default accuracy and high accuracy modes. How do the numbers compare between the two modes? How do the numbers change as you move around? The `GeoCoordinateWatcher` provides the data to pinpoint a user's location and track a user's movement. You'll extend your sample application to show the user's location and movements on an embedded map control.

13.3 Embedding a Map control

Earlier in the chapter you learned how to use the Bing Maps tasks to display maps to a user. What if you want a map control inside your application, instead of launching out to the native Bing Maps application? Have no fear: you can embed a map right inside your application.

The Silverlight `Map` control, found in the `Microsoft.Phone.Controls.Maps` namespace, performs most of the work necessary to render the map. This means you don't need to write any code to interact with the Bing Maps server to download tiles, manage zooming animations, or respond to user-initiated touch events. The `Map` control is extensible and allows the application to layer custom elements on top of the rendered map.

Your sample application displays the `Map` control in the second row of the `ContentPanel Grid` control in the LocationAndMaps project's MainPage.xaml file. As we mentioned, the `Map` control is located in the `Microsoft.Phone.Controls.Maps` namespace. This namespace isn't automatically recognized by the XAML compiler, and you need to add an `xmlns` attribute to the top of the MainPage.xaml file:

```
xmlns:maps="clr-namespace:Microsoft.Phone.Controls.Maps;
➥ assembly=Microsoft.Phone.Controls.Maps"
```

Now you can add the `Map` control to `ContentPanel`:

```
<maps:Map x:Name="mapControl" Grid.Row="1"
    ScaleVisibility="Visible" ZoomBarVisibility="Visible">
    <maps:MapLayer x:Name="routeLayer">
        <maps:MapPolyline x:Name="routeLine" Stroke="Black"
            StrokeThickness="5" />
    </maps:MapLayer>
    <maps:MapLayer x:Name="pinLayer" />
</maps:Map>
```

Drag, Pinch, and other gestures supported in the native Bing Maps application are supported in the `Map` control. You can also use the `Map`'s `ZoomBarVisibility` property to display a zoom bar allowing the user to control zooming with simple touches instead of the `Pinch` gesture. The `Map` control also allows the developer to specify the `ZoomLevel` of the map, and whether the map's current distance scale is displayed.

The `Map` control enables applications to layer custom user interface controls on top of the rendered map. The `Microsoft.Phone.Controls.Maps` namespace even provides a special layout container named `MapLayer` to simplify displaying custom widgets in the map. You've added two `MapLayers` to the `Map` control that you'll use to draw the user's position and movements. What makes `MapLayer` so special is its ability to translate geographical coordinates into screen coordinates. You'll see how useful this feature can be in the next sections when you display the device's current location with a `Pushpin`.

A `Pushpin` is just one of the mapping controls provided by the `Microsoft.Phone.Controls.Maps` namespace. The mapping library also contains the `MapPolyline` and

`MapPolygon` controls for drawing lines and shapes. In the previous snippet, you added a `MapPolyline` control to the route layer, which you'll use to draw the user's movements.

If you run the application now, you should see the `Map` control displayed in the bottom half of the phone's screen. But the device's current location isn't represented on the map. The `Map` control doesn't know how to obtain the device's current location. You'll use the location reported by the `GeoCoordinateWatcher` to add the current location to the map.

13.3.1 Mapping the current location with the GeoCoordinateWatcher

Your sample application already detects when the current position changes by subscribing to the `GeoCoordinateWatcher`'s `PositionChanged` event. Inside the `service_PositionChanged` event handler, you need to add a call to a new `UpdateMap` method:

```
UpdatePositionText(location);
UpdateMap(location);
previous = location;
```

`UpdateMap` will be responsible for drawing the user's position when the location service is first started, updating the current position as the user moves, and drawing the user's route on the map. The implementation of `UpdateMap` is shown in the following listing.

Listing 13.9 Displaying the current location on the map

```
private void UpdateMap(GeoCoordinate location)
{
    if (previous.IsUnknown)
    {
        Pushpin startPin = CreatePushpin(location, Colors.Green);
        routeLayer.AddChild(startPin, startPin.Location);
    }

    pinLayer.Children.Clear();
    Pushpin pin = CreatePushpin(location, Colors.Blue, "You");
    pinLayer.AddChild(pin, pin.Location);

    routeLine.Locations.Add(location);

    mapControl.Center = location;
    mapControl.SetView(
        LocationRect.CreateLocationRect(routeLine.Locations));
}
```

Show starting location ❶

Show current location ❷

❸ **Center and zoom map**

You start by adding a new `Pushpin` control to the map's route layer ❶ the first time the `PositionChanged` event is raised. You add another `Pushpin` control to represent the current position of the user ❷. Note that you remove any existing `Pushpins` in the pin layer first. Next you add another point drawn by the `MapPolyline` control that displays the user's movements. The `Map` control provides a `Center` property that's used to set the `GeoCoordinate` displayed in the center of the screen. When you set the `Center` property ❸, the `Map` control will automatically scroll the specified location into view, using the appropriate zoom and scroll animations. Finally, you instruct the `Map` control to zoom and scroll the route into view by calling the `SetView` method.

A `MapPolyline` knows how to translate a `GeoCoordinate` into screen coordinates. Points are added to the `MapPolyline` by adding a `GeoCoordinate` to the poly line's `Locations` collection. The `MapPolyline` will then render a line segment between each point.

The push pins are built in a method named `CreatePushpin`. Let's look at what a push pin is, and how the Maps library makes it easy to create one.

13.3.2 Creating a push pin

The `Microsoft.Phone.Controls.Maps` namespace includes a `Pushpin` control that simplifies the process of labeling points on the map. The `Pushpin` control uses a `Location` property to identify exactly where the push pin should be located on the map. Figure 13.4 shows a map with two different push pins.

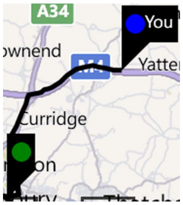


Figure 13.4 Two `Pushpins`, one containing an `Ellipse` and the other containing both an `Ellipse` and a `TextBlock`

The `Pushpin` control is derived from `ContentControl`, allowing a developer to add other Silverlight elements to the `Pushpin`, such as `Ellipses` and `TextBlocks`. The developer could also replace the entire `ContentTemplate` if the default `Pushpin` look and feel is undesirable. Adding custom content suffices for this sample application. The next listing shows how you create a `Pushpin` with custom content in the `CreatePushpin` method.

Listing 13.10 Creating a `Pushpin`

```
private Pushpin CreatePushpin(GeoCoordinate location, Color color,
    string label = null)
{
    StackPanel content = new StackPanel { Height = 30,
        Orientation = System.Windows.Controls.Orientation.Horizontal };

    content.Children.Add(new Ellipse {
        Height = 20, Width = 20,
        Fill = new SolidColorBrush(color) });

    content.Children.Add(new TextBlock { Text = label });

    return new Pushpin { Location = location,
        Content = content };
}
```

① **Stack Ellipse next to TextBlock**

② **Add custom content**

You start by creating a `StackPanel` to stack an `Ellipse` side by side with a `TextBlock` ①. Next you add a 20 x 20 pixel `Ellipse` with its `Fill` property set to a `SolidColorBrush` that uses the specified color. You also create `TextBlock` to display the text specified in the label parameter. Finally, you construct a new `Pushpin` with the specified `Location` and your `StackPanel` ②.

Run the application again, start the location service, and this time you should see a map of the world with your location identified with a push pin. If you move about, you should see the map update the current position and draw your route. The other thing you probably noticed is a watermark complaining about invalid credentials.

Behind the scenes, the `Map` control is making calls to the Bing Maps web services. These web services require a valid developer or application key. If you have a valid developer key, use it to assign the `Map` control's `CredentialsProvider` property. If you don't have a valid developer key, consider signing up for one, as you'll need it to finish the sample application.

The fourth and last feature of the sample application uses the Bing Maps web services.

13.4 Using the Bing Maps Services

The last feature of the LocationAndMaps application displays an address for the user's current position. The address is displayed in a pop-up `MessageBox`. Though not a very elegant user interface, the intention here is to demonstrate how to use the Bing Maps web services in your location-aware application.

Bing Maps Services provide a number of different APIs exposing a variety of data. There are APIs for converting an address into latitude and longitude (geocoding) and from latitude and longitude into an address (reverse geocoding). There are APIs that return map images in various formats. You can even obtain driving directions and traffic data. The web services present both SOAP and REST APIs.

Before you can use the Bing Maps web services, you need to create an account and request a developer license. To request a developer key, visit the website http://www.microsoft.com/maps/ and look for the *Get an Account* link. After logging in with a Windows Live ID, you'll be prompted for various bits of information such as your name, company name, and website. At the end of the process you'll be presented with a Bing Maps key, which is a long string of alphanumeric characters and symbols.

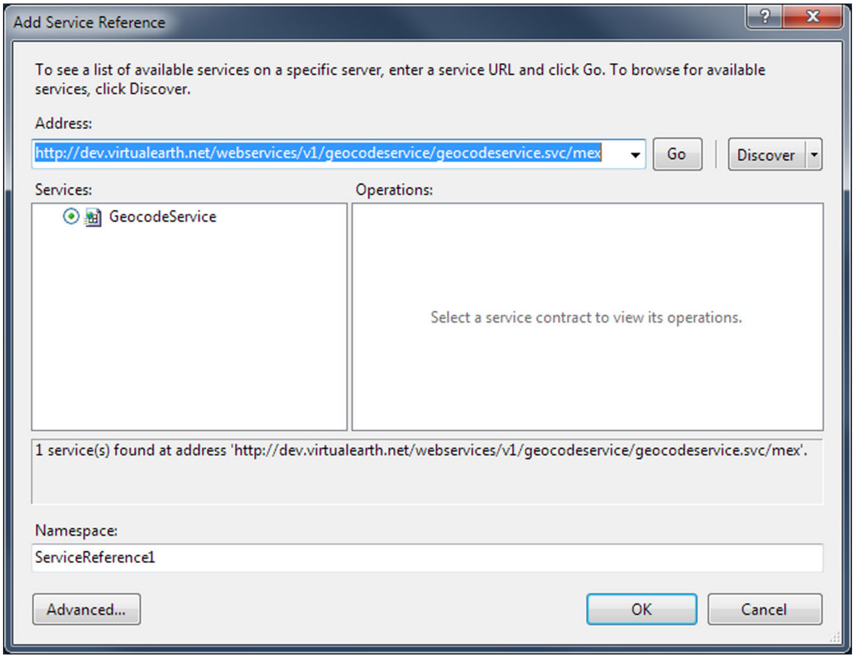
> **NOTE** Many of the APIs and features provided by the Bing Maps Services can be used free of charge by mobile-based applications. Use of certain APIs may result in charges and fees. Consult the Bing Maps documentation or send an email to the Bing Maps licensing team at maplic@microsoft.com for specifics.

Once you have your developer key, you can use it as the value assigned to the `CredentialsProvider` property of the `Map` control. You'll also use the key when sending requests to the web services. Requests that fail to include the key are rejected by the web services.

The LocationAndMaps sample application uses the SOAP-based web services. The Windows Phone developer tools include tools to easily add service references and generate client-side classes that simplify web service communication.

13.4.1 Adding the service reference

The LocationAndMaps sample application will use just one of the Bing Maps web services. You'll use the Geocode Service to transform a longitude and latitude into a street



To see a list of available services on a specific server, enter a service URL and click Go. To browse for available services, click Discover.

Address: http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc/mex

GeocodeService

Select a service contract to view its operations.

1 service(s) found at address 'http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc/mex'.

Namespace: ServiceReference1

Figure 13.5 Adding a service reference for the Geocode Service to the PhoneMaps sample application

address. Before you can add a service reference, you need to know the SOAP URL of the Geocode Service, http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc.

Adding service references is as easy as selecting the Add Service Reference menu item from Visual Studio's Project menu. When the Add Service Reference dialog appears, type the SOAP URL into the Address input control and press the Go button. Change the default value in the Namespace input control to use a value more suited to your naming conventions. When you press OK, the new service will appear under the Service References folder in the Solution Explorer. Figure 13.5 shows the add dialog for importing the Geocode Service.

You're almost ready to start making calls to the Bing Maps web services to convert the user's location to a street address. Converting latitude and longitude to a street address is a process named reverse geocoding.

13.4.2 Reverse geocoding

The Geocode web service defines several classes and enumerations that allow geographical coordinates to be translated into a street address and vice versa. The street address associated to a location can be retrieved using the `GeocodeServiceClient`, which was generated during the Add Service References process.

The Geocode web service will be called when the user taps the geocode application bar button you added when you created the sample application. At that time you added an empty `Click` event handler for the button, named `geocode_Click`. You'll update the event handler to call the Geocode service.

Before calling the Geocode service, you must create and populate a `ReverseGeocodeRequest`. The request object is populated with the developer key and by setting the `Location` property. The following listing demonstrates how you create and send a `ReverseGeocodeRequest` in the button's `Click` event handler.

Listing 13.11 Sending a Geocode request

```
private void geocode_Click(object sender, EventArgs e)
{
    GeocodeServiceClient geocodeService =
        new GeocodeServiceClient("BasicHttpBinding_IGeocodeService");
    geocodeService.ReverseGeocodeCompleted +=
        geocodeService_ReverseGeocodeCompleted;

    ReverseGeocodeRequest geocodeRequest = new ReverseGeocodeRequest()
    {
        Credentials = new Credentials
        {
            ApplicationId = "your developer key here"
        },
        Location = new GeocodeLocation
        {
            Latitude = previous.Latitude,
            Longitude = previous.Longitude,
        }
    };
    geocodeService.ReverseGeocodeAsync(geocodeRequest);
}
```

① **Enter developer key**

② **Send async requests**

First you construct a `GeocodeServiceClient` and subscribe to the `ReverseGeocodeCompleted` event. All calls to SOAP services are executed asynchronously. The completed event handler will be called once the web service data is returned to your client. Next you construct a `ReverseGeocodeRequest` complete with credentials ①. Assign your Bing Maps key to the `ApplicationId` property of the `Credentials` class. You also fill in the `Location` property using the longitude and latitude stored in the `previous` field. Finally, you submit the request to the Geocode Service ②.

When the web service call completes, your completed event handler is called. The next listing shows the implementation of the event hander named `geocodeService_ReverseGeocodeCompleted`.

Listing 13.12 Handling the completed event

```
void geocodeService_ReverseGeocodeCompleted(object sender,
    ReverseGeocodeCompletedEventArgs e)
{
    if (e.Cancelled || e.Error != null ||
```

① **Check for web service errors**

```
       e.Result.ResponseSummary.StatusCode != ResponseStatusCode.Success)
    {
        MessageBox.Show("Unable to complete the ReverseGeocode request");
        return;
    }
    GeocodeResponse response = e.Result;
    if (response.Results.Count > 0)
    {
        GeocodeResult address = response.Results[0];
        MessageBox.Show(address.DisplayName);
    }
}
```

② **Display first address**

The completed event handler will be called even when an error occurs during the web service call. When an error occurs, you display a message to the user and return ①. Next you read the `GeocodeResponse` from the event args and check whether at least one `GeocodeResult` was returned by the service. You then show the user the value of the `GeocodeResult`'s `DisplayName` property in a `MessageBox` ②.

> **NOTE** The `GeocodeResponse` object may contain more than one result. A more robust application might display all possible results. Your simple application uses only the first result in the list.

The `GeocodeResult` class exposes several properties providing details about the location referenced by the result. The `Address` property will give you the physical address of the location. The `EntityType` property will tell you whether the location is an airport, a bridge, a shipwreck, or one of approximately 200 other types of entities.

Run the sample application one last time and start the location service. Sometime after the location service returns the current position, tap the geocode button. After a slight delay you should see an address displayed in pop-up.

We've shown you a couple of different ways to integrate an application with Bing Maps. You can use the two launcher tasks to display the native Bing Maps application outside of your application. We also showed you how to use the Silverlight `Map` Control to host the Bing Maps inside you application. In the next section we show how to use the `WebBrowserTask` to launch Internet Explorer 9 outside your application. We also show you how to use the Silverlight `WebBrowser` control to host an application written almost entirely in HTML 5 and JavaScript.

13.5 Building an HTML 5-based application

The release of Windows Phone 7.5 was the first time mobile phones were able to use Internet Explorer 9. Internet Explorer 9 for mobile includes several new features such as hardware acceleration and support for HTML 5, ECMAScript 5, CSS3, and Scalable Vector Graphics (SVG). HTML 5 includes several new tags and features to support web storage, geolocation, audio, and video.

> **NOTE** This chapter isn't intended to be a full introduction to HTML 5 and related technologies. Instead we want to show you how you can create an

HTML 5 with JavaScript application that leverages a few of the HTML 5 features. For a complete introduction to HTML 5 consider reading *Hello! HTML5 and CSS3* by Rob Crowther. You can view more details about the book on the Manning Publications website at http://www.manning.com/crowther.

Throughout the remainder of the chapter we're going to look at how Windows Phone applications can leverage Internet Explorer 9. We're going to start by showing you how to use the `WebBrowserTask` launcher to start Internet Explorer from within an application. We're then going to show you how to embed a browser inside of a Silverlight application. The sample application you're going to build, shown in figure 13.6, will use HTML and JavaScript to render the user interface.

The Html5App sample application runs from HTML files included in the application's .xap file using the Silverlight `WebBrowser` control. The browser loads the files directly from isolated storage. We also show you JavaScript running inside the browser control that interoperates with C# application code, a technique that can be leveraged to expose Windows Phone APIs to an HTML with JavaScript application.

You create the Html5App project in the next section. The first feature of the sample application shows you how to use Internet Explorer from within a Silverlight application.

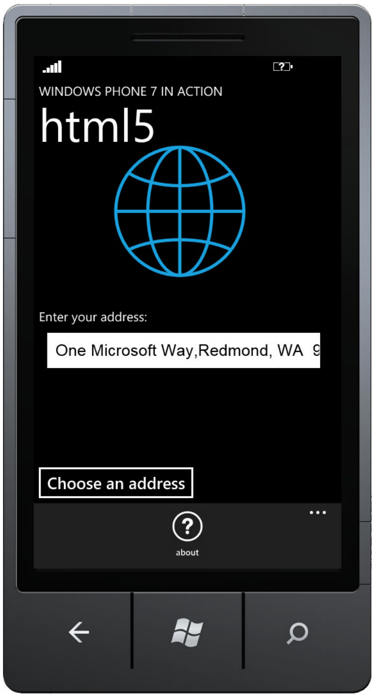


Figure 13.6 The HTML 5 sample application

13.5.1 Launching Internet Explorer

Internet Explorer can be launched from any third-party application using the `WebBrowserTask` launcher class. You learned how to use launchers and choosers in chapter 4. The `WebBrowserTask` launcher exposes a single property named `Uri`, which accepts a standard `Uri` object. We show you how to use the `WebBrowserTask` when a user taps an application bar button in your new sample application.

Open Visual Studio and create a new project named Html5App using the Windows Phone Application project template. Open up MainPage.xaml and add an `ApplicationBarIconButton` as shown in the following code snippet:

```
<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" Mode="Minimized">
        <shell:ApplicationBarIconButton  Text="about" Click="about_Click"
            IconUri="/Images/appbar.questionmark.rest.png" />
    </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

You use an image from the Windows Phone SDK icon library that was added to a project folder named Images. Internet Explorer is launched from the button's `Click` event handler:

```
private void about_Click(object sender, EventArgs e)
{
    var task = new WebBrowserTask
    {
        Uri = new Uri("http://www.manning.com/perga", UriKind.Absolute)
    };
    task.Show();
}
```

When you run the application and tap the About button, Internet Explorer is launched. Assuming the device has a network connection, the browser will load the book's page on the Manning website. The drawback to using the `WebBrowserTask` is that the application is pushed to the background and the user leaves the application. To keep the user from leaving your application, Silverlight includes the `WebBrowser` control, which provides developers a way to embed Internet Explorer right inside an application.

13.5.2 Embedding Internet Explorer

The `WebBrowser` class is found in the `Microsoft.Phone.Controls` namespace. The `WebBrowser` class has several properties, events, and methods that we'll explore as you continue to build the Hmtl5Application. You're going to jump right into using the browser by adding one to MainPage.xaml.

Other than the application bar you've already built, the entire user interface of the Html5Application will be built using HTML. You do not need the XAML markup for the standard application and page or the `ContentPanel` grid. Delete the contents of the `RootLayout` grid and add a `WebBrowser` control:

```
<Grid x:Name="LayoutRoot" Background="Transparent">
    <phone:WebBrowser x:Name="webBrowser"
        Source="http://www.manning.com/perga/" />
</Grid>
```

The `WebBrowser` supports loading web pages using three different APIs. In the preceding snippet, you specify the address of the web page to display in the `WebBrowser` control using the `Source` property. The `WebBrowser` control also provides the `Navigate` and `NavigateToString` methods. The `Navigate` method performs the same operation as the `Source` property by loading the pages specified in an `Uri` instance. The `NavigateToString` displays the HTML markup contained in the string passed to the method. You'll use both forms of the `Navigate` method later in the chapter.

When loading a web page, the `WebBrowser` control raises three different events during normal navigation:

- `Navigating`–Raised when the page is loading
- `Navigated`—Raised when navigation to the page has succeeded
- `LoadCompleted`—Raised when page has been loaded

Each of the `WebBrowser` navigation events might be called more than once when navigating to a web page. The number of events raised depends on the content of the page. Redirects and frames are a couple of the elements that may cause multiple navigation events to be raised for a page. The `Navigating` event has a particularly useful function, because you can interrupt the navigation by setting to the `Cancel` property to `true` while handling the event.

`NavigationFailed` is another event raised by the `WebBrowser`. As you might suspect, the `NavigationFailed` event is raised when the browser is unable to load a page. If your code doesn't handle `NavigationFailed`, the `WebBrowser` control will display an error message of one form or another, depending on the type of error.

The `WebBrowser` control works nicely with content served up from a website. The `WebBrowser` can also load web pages from files in an application's isolated storage. Let's look at how to add HTML files to your sample application project.

13.5.3 Adding HTML pages to the project

The `WebBrowser` control can load a page that exists in an application's isolated storage. You're going to add HTML and CSS files to the Html5Application project, copy the files to isolated storage, and use the files as the application's user interface. You'll start by adding a new folder to the project named Html and adding two text files to the Html folder. The files should be named Html5App.css and MainPage.html. The build action property should be set to `Content` for both of the new web files.

The next step is to update MainPage.xaml and its code behind file to support navigating to MainPage.html during startup. Start by updating the `WebBrowser` control:

```
<phone:WebBrowser x:Name="webBrowser" Base="Html" IsScriptEnabled="True"
    NavigationFailed="webBrowser_NavigationFailed" />
```

You're no longer setting the `Source` property and are now setting the `Base` and `IsScriptEnabled` properties. The `Base` property tells the browser which folder in isolated storage is the root web application folder. We'll show you how to create the Html folder in isolated storage in the next few pages. By default, the `WebBrowser` control disables JavaScript execution, so you set `IsScriptEnabled` to allow you to create dynamic web pages.

The `IsScriptEnabled` property isn't applied to the currently loaded page. This means that you need to load your application pages after setting the `IsScriptEnabled` to `true`. Open MainPage.xaml.cs and tell the `WebBrowser` to navigate to MainPage.html inside the constructor:

```
public MainPage()
{
    InitializeComponent();
    webBrowser.Navigate(new Uri("MainPage.html", UriKind.Relative));
}
```

You set the `UriKind` to `Relative` as a hint to the browser that it should look in isolated storage for the web page.

When building mobile web applications, you should consider the browser viewport. By default, IE9 sets the viewport width to 1024 and adjusts the viewport height big enough to contain the page when rendered at 1024. The viewport is then scaled so that the entire viewport fits in the browser window. The `meta` tag is used to set the viewport's width:

```
<meta name="viewport" content="width=device-width" />
```

In this case, the viewport width has been set to the device's default width. On Windows Phone 7, the default `device-width` isn't the actual screen resolution of 480px, but is 320px to support backward compatibility.

The Html5Application's main page, shown in the following listing, looks a lot like the main page of the Hello World application you built way back in chapter 2. The page has the standard Metro application and page titles and asks the user to input a name.

Listing 13.13 HTML5 for the applications main page

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport"
        content="width=480, height=768, user-scalable=no" />
    <link rel="stylesheet" href="Html5App.css" />
    <title>Windows Phone 7 in Action</title>
</head>
<body>
    <header>
        <h2>WINDOWS PHONE 7 IN ACTION</h2>
        <h1>html5</h1>
    </header>
    <section>
        <label for="address">Enter your address:</label><br />
        <input id="address" /><br />
    </section>
    <footer>
        <nav>
            <button type="button" class="wp7button">
              Choose an address
            </button>
        </nav>
    </footer>
</body>
</html>
```

① **Set viewport size**

Metro page titles

② **Styled to look like button**

MainPage.html is a standard HTML 5 page, complete with a simplified `DOCTYPE`, and uses the new `header`, `section`, and `footer` elements. The viewport `meta` tag ① is used to set the width and height of the browser viewport. The `user-scalable` value is set to *no* to prevent the user from zooming the page. In the footer you add a `button` ② that, when clicked, will launch an `AddressChooserTask`. You add the CSS class named `wp7Button` to the button to allow you to style the link to look like a Metro button.

The Hello World application from chapter 2 used Silverlight drawing primitives from the `System.Windows.Shapes` namespace to display a globe. Internet Explorer 9 supports SVG elements that are similar to Silverlight's `Shape` classes. The next listing demonstrates how to use SVG to replicate the Hello World globe.

Listing 13.14 SVG markup for a globe

```
<section>
    <svg xmlns="http://www.w3.org/2000/svg" version="1.1">
        <ellipse cx="100" cy="100" rx="100" ry="100" />
        <ellipse cx="100" cy="100" rx="50" ry="100" />
        <path d="M100 0 L100 200" />
        <path d="M0 100 L200 100" />
        <path d="M20 40 A100 50 0 0 0 180 40" />
        <path d="M20 160 A110 50 0 0 1 180 160" />
    </svg>
</section>
```

Add the section containing the globe immediately following the header element in MainPage.html. The sphere of the globe and the two arced meridians are drawn with `ellipses`. The straight meridian and the three parallels are drawn with `paths`.

In order to display the globe properly, you need to apply some CSS. Add a `style` element to the document's `head` element:

```
<style>
    ellipse, path{
        fill: transparent; stroke: #1BA1E2; stroke-width: 5px;
    }
    svg{
        margin-left: 120px; margin-right: 160px;
    }
</style>
```

The style specifies a transparent fill and 5-pixel-wide blue stroke. The color value used matches the color used in Metro's blue theme. You can also style other elements on the page to match the style specified in the Metro design language.

13.5.4 Matching the Metro style

If you're building a JavaScript/HTML application to look like a native application, you should consider trying to match Metro's look and feel. How can you build a CSS to match Metro? What colors, fonts, and sizes should be used? You can find this information in the Themes.xaml and System.Windows.xaml files included in the Windows Phone 7.1 SDK.

Let's look at a couple of styles defined in Html5App.css used to make the Html5Application match the Metro style. The `body` style is defined to use the Segoe font, with a default size of 20px:

```
body{
  font-family: Segoe WP; font-size: 20px; background: black; color: white;
}
```

The `body` style also sets the background to black and the foreground to white to match Metro's Dark theme.

You also want to define a custom style to make input buttons look like native Silverlight buttons:

```
.wp7button{
  color: white; background: transparent; text-decoration: none;
  border-style: solid; border-color: white; border-width: 3px;
  font-size:25.33px; font-family:Segoe WP Semibold;
  padding-left: 10px; padding-top: 3px;
  padding-right: 10px; padding-bottom: 5px;
}
```

The `wp7button` style uses the Segoe WP Semibold font, a solid white border, and a transparent background. We've created styles for several other elements, which you can find in the Html5App.css file provided with the book's sample source code.

If you try to run the application now, you'll be disappointed with the results. Your web files are included in the application's .xap package and placed in the appdata folder, but aren't automatically copied to isolated storage. The WebBrowser can only read local files from isolated storage.

13.5.5 Working from Isolated Storage

In order to work around the `WebBrowser` control's inability to load files from the appdata folder, you need to copy the application's web files into isolated storage. Remember that you set the `WebBrowser`'s `Base` property to Html, meaning you need to copy the project files into a folder named Html that you create in isolated storage, as shown in figure 13.7. You need to perform the copy operation before MainPage.xmal is loaded.

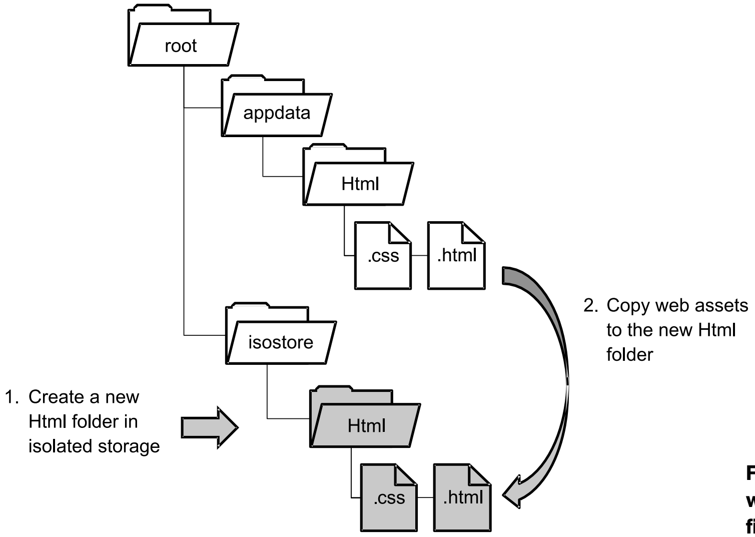

root
appdata
Html
.css
.html
isostore
Html
.css
.html
1. Create a new Html folder in isolated storage
2. Copy web assets to the new Html folder

Figure 13.7 Copying web assets from the .xap file into isolated storage

The `Launching` event handler in App.xaml.cs is the ideal place for your copy files' code. In chapter 3, you learned that the `Launching` event is raised when an application instance is first started, but isn't called when a dormant application is restarted. The following listing shows the implementation for copying the web files to isolated storage.

Listing 13.15 Copy application files to isolated storage

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    string[] applicationFiles = {  @"Html\Html5App.css",
        @"Html\MainPage.html" };

    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        storage.CreateDirectory("Html");

        foreach (string file in applicationFiles)
        {
            StreamResourceInfo sourceInfo = Application
                .GetResourceStream(new Uri(file, UriKind.Relative));
            using (var source = sourceInfo.Stream)
            {
                using (IsolatedStorageFileStream target =
                    storage.CreateFile(file))
                {
                    byte[] buffer = new byte[4096];
                    int bytesRead = -1;

                    while ((bytesRead =
                        source.Read(buffer, 0, buffer.Length)) > 0)
                    {
                        target.Write(buffer, 0, bytesRead);
                    }
                }
            }
        }
    }
}
```

❶ **List of files to copy**

❷ **Create Html folder**

❸ **Open file in appdata folder**

You start by declaring each file ❶ that needs to be copied from the appdata folder into isolated storage. Next you open the storage container and create the Html folder ❷. You then loop through the file list, opening each file ❸ and copying the bytes into a new file you create in isolated storage. You can read more about isolated storage file operations in chapter 5.

Now that the web application files are in isolated storage, you can load MainPage .html into the web browser. Run the application and you should see a friendly Hello World page written in HTML 5. But nothing happens when you tap the Choose an Address button. You'd like the button to launch the `AddressChooserTask`. How do you access a native API that's not automatically exposed to JavaScript? Fortunately the

`WebBrowser` allows you to bridge JavaScript and C# to access almost any API in the Windows Phone SDK.

13.5.6 Bridging C# and JavaScript

As you already know, the Windows Phone SDK includes several launchers and choosers that developers can use to interact with native applications. You've seen three examples in this chapter. You first learned about launchers and choosers in chapter 4, and have used them throughout the book. Using features of the `WebBrowser` control, you're going to learn how you can use the `AddressChooserTask` to display an address in the Location.html web page.

The two features of the `WebBrowser` control that enable C# to interoperate with JavaScript are the `ScriptNotify` event and the `InvokeScript` method. As seen in figure 3.8, the workflow started by a user clicking on an HTML button flows through the `ScriptNotify` event handler, where you show the `AddressChooserTask`. When the task is complete, code execution flows back to the web page through the `InvokeScript` method.

Wire up the `ScriptNotify` event of the browser control you're using in MainPage.xaml:

```
<phone:WebBrowser x:Name="webBrowser" Base="Html" IsScriptEnabled="True"
    NavigationFailed="webBrowser_NavigationFailed"
    IsGeolocationEnabled="True" ScriptNotify="webBrowser_ScriptNotify" />
```

The `WebBrowser` adds a JavaScript function named notify to the browser DOM. When JavaScript code calls the notify function, the `ScriptNotify` event is raised. You'll see this in action by adding an `onclick` event handler to the Choose Address HTML 5 button in the Location.html page:

```
<button type="button" class="wp7button"
    onclick="window.external.Notify('chooseAddress');" >
    Choose an address</button>
```

Inside the click event handler, you call the `Notify` function, passing a hint to the C# code informing it that you want the `AddressChooserTask` launched. The `Notify` function

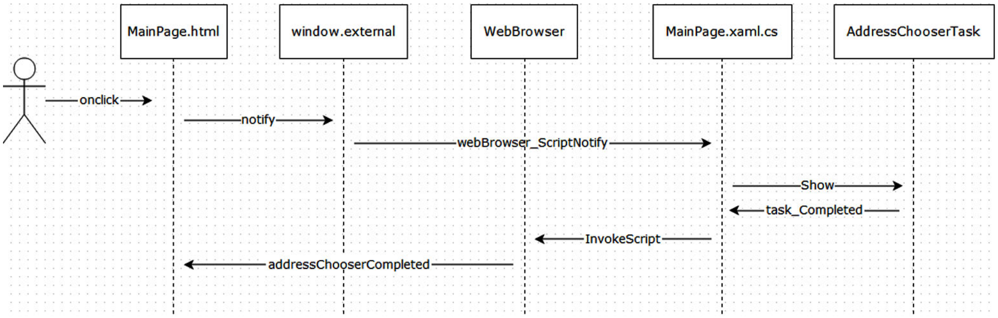

MainPage.html
window.external
WebBrowser
MainPage.xaml.cs
AddressChooserTask
onclick
notify
webBrowser_ScriptNotify
Show
task_Completed
InvokeScript
addressChooserCompleted

Figure 13.8 A sequence diagram showing the code execution from JavaScript to C# and back

accepts a single string. When the `Notify` function is called, the `WebBrowser` creates an instance of the `NotifyEventArgs` class and sets its `Value` property to the string sent to the `Notify` function. The `WebBrowser` then raises the `ScriptNotify` event, passing the `NotifyEventArgs` instance to the event handler.

Inside your implementation of the event handler, you examine the `Value` property for the string *chooseAddress*, and if you find it, you instantiate an `AddressChooserTask`. You subscribe to the task's `Completed` event and launch the chooser by calling the `Show` method:

```
private void webBrowser_ScriptNotify(object sender, NotifyEventArgs e)
{
    if (e.Value == "chooseAddress")
    {
        var task = new AddressChooserTask();
        task.Completed += task_Completed;
        task.Show();
    }
}
```

When the address chooser returns, it may have a valid address, but it may have an error. The user may have canceled the task by pressing the Back button without selecting an address. You need to account for each of these scenarios in your `Completed` event handler:

```
void task_Completed(object sender, AddressResult e)
{
    string message;
    if (e.Error != null || e.TaskResult != TaskResult.OK)
        message = "No address chosen";
    else
        message = e.Address.Replace("\r\n", ",");
    webBrowser.InvokeScript("addressChooserCompleted", message);
}
```

The `InvokeScript` method accepts the name of the JavaScript method to invoke, and zero or more additional string parameters that will be passed along to the specified function. In this sample application, you call a function named `addressChooserCompleted` with either an error message or the value of the `AddressResult`'s `Address` property passed to the task completed event handler.

The last thing you need to do is create the JavaScript function named `addressChooserCompleted` and add a script block to MainPage.html. The script block should be entered after the closing `body` tag:

```
<script>
function addressChooserCompleted(address) {
    document.getElementById('address').innerText = address;
}
</script>
```

The function implementation copies the provided address, which might be an error message, into the `innerText` property of the address input control.

Now that you're displaying an address chosen by the user, you've completed your HTML 5 sample application. Run the application and experiment with choosing an address.

13.6 Summary

In this chapter you learned how to use a few launchers and a couple of Silverlight controls to integrate Bing Maps and Internet Explorer into an application. The `BingMapsTask` and `BingMapsDirectionsTask` launch the Bing Maps application from your code. The Silverlight Map control enables you to embed maps inside your application. The `WebBrowserTask` can be used to launch Internet Explorer 9, whereas the Silverlight `WebBrowser` control allows you to embed IE9 into your application.

The `GeoCoordinateWatcher`, also known as the Location Service, uses data from the GPS, cellular network, wireless access points, and a web service to report the phone's longitude and latitude. The `GeoCoordinateWatcher` can optionally report additional information such as altitude, speed, and heading.

TIP If you're interested in location-based and mapping technologies, we recommend reading the book *Location-Aware Applications* by Richard Ferraro and Murat Aktihanoglu. You can learn more about the book on Manning's website at http://www.manning.com/ferraro.

In addition to the Bing Maps tasks and the `Map` control, we looked at how to use the Bing Maps Web Services. You learned how to use one of the available Bing Maps APIs for geocoding. Bing Maps provides other services such as route mapping, reverse geocoding, traffic information, and map imagery. Microsoft isn't the only organization that provides mapping and location web services. Yahoo, Google, and several other organizations provide free and fee-based mapping services.

We showed you a few techniques that leverage the `WebBrowser` and allow you to build HTML 5 plus JavaScript applications. If you decide to build HTML 5 plus JavaScript applications, you might consider using third-party frameworks like Apache Cordova (a.k.a PhoneGap). Apache Cordova is an HTML 5 framework that allows you to build native applications for Windows Phone 7 and several other mobile devices using the same code base.

In the next chapter we introduce you to 3D graphics applications and how to mix Silverlight with 3D XNA development. We'll give you a crash course in the XNA Graphics Framework and how to build a game where a player wanders around a virtual 3D world.

Part 4

Silverlight and the XNA Framework

Windows Phone 7 is an exciting platform for both application and game development. The final part of this book demonstrates how Silverlight and the XNA Framework can be used together to build exciting games and applications. The XNA Framework includes a rich library for three-dimensional modeling and rendering. Three-dimensional models have been used for years in drafting applications and simulations of real-world environments such as buildings, open space, landscaping, and maps.

In chapter 14, you'll use the Windows Phone Silverlight and XNA Application template to create a Hello World game, and learn the techniques used to render Silverlight user interface elements with the XNA graphics framework. We give you a crash course in XNA concepts such as 3D animation techniques, collision detection, and the game loop.

You continue working with the sample game in chapter 15, where you learn how to use sprites for 2D graphics and animation. You'll use raw touches, gestures, the motion sensor, and the Mouse API to let a game player wander around the game world. We also show you how to integrate a Silverlight Button control into the game.

14

Integrating Silverlight with XNA

This chapter covers

- Creating a Silverlight and XNA application
- Working with the XNA Framework
- Collision detection
- Rendering Silverlight controls with XNA

You're a Silverlight developer working on business and consumer applications. The XNA Framework is used to build games, so you might be asking yourself why you should read this chapter. Games don't have exclusive rights to three-dimensional modeling and rendering. Three-dimensional models have been used for years in drafting applications. Three-dimensional models are ideal for simulations of real-world environments such as buildings, open space, landscaping, and maps. We encourage you to read through these last two chapters and introduce yourself to the features the XNA Framework has to offer, whether or not you're building games.

Silverlight has a powerful animation library and support for pseudo-3D graphics through various perspective transforms. When your visualization requirements exceed the from/to/by or key-frame animation techniques supported by Silverlight, or require complex 3D effects, you should consider using XNA graphics. Real-world examples include modeling buildings for real estate applications, complex visualization techniques for data analysis or reporting applications, or even fun augmented reality applications that blend input from the camera with information from social networking applications. In this chapter and the next, we show you how to make use of the 2D and 3D graphics library provided by the XNA Framework.

The XNA Framework provides more than just graphics. In the next chapter, we show you how to use the Touch API to work with gestures and raw multi-point touch information. In this chapter and the next, we use the vector, matrix, and bounding shape libraries included in the XNA Framework to work with 2D and 3D coordinate systems. Although it's not part of the XNA Framework, we also show you how to use the `VibrateController` to shake the phone.

Since 1995, the Windows operating system has included DirectX libraries for building computer games. The DirectX libraries allow low-level access to a computer's audio, video, and input hardware. Games built using DirectX are written in C, C++, or other unmanaged programming languages. Shortly after the release of the first version of the .NET Framework, Microsoft introduced *Managed DirectX*, a managed library that abstracted the low-level DirectX APIs and provided C# developers an option for building games for Windows computers.

When Microsoft replaced the original Xbox with the Xbox 360, they also replaced Managed DirectX with the XNA Game Studio and the XNA Framework. Microsoft created XNA Game Studio to simplify DirectX game development for students, hobbyists, and independent game developers. From the beginning, game programmers could use XNA to build games for both the Windows PC and the Xbox from the same code base. Support for Zune media players was added with Game Studio 3.0, and version 3.1 introduced touch screen support for the Zune HD. Game Studio 4.0 was released concurrently with the Windows Phone Developer Tools and is used to create games for Windows Phone 7.

One drawback to the XNA Framework is the lack of a control library. There are no buttons or text boxes or any of the other widgets you can find in the `System.Windows.Controls` namespace. XNA developers often waste a good deal of time implementing simple user interface controls instead of focusing on actual game development. Beginning with Windows Phone SDK 7.1, game developers can combine the rich control library in Silverlight with the rich graphics libraries in the XNA Framework. Applications that combine Silverlight and XNA graphics are called *rich graphics* or *3D graphics applications*.

Silverlight and XNA can't both draw to the GPU at the same time. 3D Graphics applications are made possible due to new code in the Silverlight Framework that enables Silverlight components to defer drawing. Instead of drawing to the GPU, Silverlight components in 3D graphics applications render to a XNA texture, which the XNA Framework then draws to the screen.

This chapter introduces 3D graphics applications and how to mix Silverlight with 3D XNA development. We'll give you a crash course on the XNA Framework and how to build a game where a player wanders around a virtual 3D world. The game that you'll create is shown in figure 14.1.

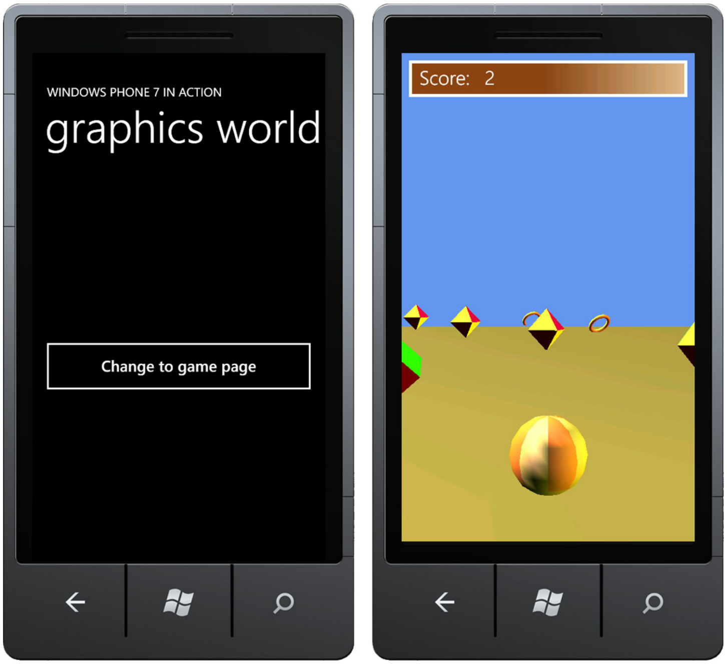


Figure 14.1 The sample 3D graphics application built in this chapter

The game consists of a 3D map with several shapes scattered across the ground. As the player moves around and collides with the shapes, they'll disappear from the map. In the next chapter we show you how to move around using the touch screen and the motion sensor. In this chapter, movement will be implemented with a hard-coded script. You're going to get started by creating a new Silverlight and XNA Application project.

14.1 Creating a Silverlight with XNA application

The Windows Phone 7.1 Developer Tools include a project template for creating integrated Silverlight and XNA applications. The project template named Windows Phone Silverlight and XNA Application appears in both the Silverlight for Windows Phone category and the XNA Game Studio category.

Open Visual Studio, select New Project from the file menu, choose the Windows Phone Silverlight and XNA Application template, and name the project GraphicsWorld. Once you click OK, a new Visual Studio solution is created with a Silverlight project, an XNA class library named GraphicsWorldLib, and a standalone content project named GraphicsWorldLibContent.

Many of the third-party tools used to create game assets such as sounds, images, or 3D models don't export directly to XNA runtime formats. Content projects provide the glue to compile game assets from their native formats into .xnb files, a format that XNA can consume at runtime. This removes the burden of translation from the artists and allows developers and artists to work together smoothly.

The GraphicsWorld project structure is similar to a regular Silverlight project with Properties and Reference folders, App.xaml, MainPage.xaml, GamePage.xaml, AssemblyInfo.cs, and other common files. When you run the application, you'll see a regular Silverlight page with a single button. When you click the button, you'll be navigated to GamePage.xaml and will see an empty blue screen. `MainPage` is rendered using the normal Silverlight library. Even though it's a `PhoneApplicationPage`, `GamePage` is rendered using the XNA graphics library. Because the two libraries can't both draw at the same time, they must share the graphics device.

14.1.1 Sharing the graphics device

How is the `GamePage` different from any other Silverlight page? The `GamePage` uses the XNA `GraphicsDevice` class to perform drawing operations. If you open up GamePage.xaml.cs and look for a method named `OnDraw`, you'll see the following lines of code that were generated by the project template:

```
private void OnDraw(object sender, GameTimerEventArgs e)
{
    SharedGraphicsDeviceManager.Current.GraphicsDevice.Clear(
        Color. CornflowerBlue);

    // TODO: Add your drawing code here
}
```

The `OnDraw` method is called by the `GameTimer`, which we'll discuss in the next section. The generated implementation of `OnDraw` clears the screen, setting every pixel to the color `CornflowerBlue`. Let's take a closer look at the `SharedGraphicsDeviceManager` class.

`SharedGraphicsDeviceManager` helps Silverlight and XNA share the GPU. The `SharedGraphicsDeviceManager` is an `ApplicationLifetimeObject`. This means that the `Application` class has the responsibility of creating a single instance of `SharedGraphicsDeviceManager` when the application is constructed. The `Application` class also stops and cleans up all application lifetime objects when it exits. The `SharedGraphicsDeviceManager` is declared in App.xaml along with other `ApplicationLifetimeObjects`:

```
<Application.ApplicationLifetimeObjects>
    ...
    <!--The SharedGraphicsDeviceManager is used to render with
        the XNA Graphics APIs-->
    <xna:SharedGraphicsDeviceManager />
</Application.ApplicationLifetimeObjects>
```

The Windows Phone Silverlight and XNA Application template automatically added the `SharedGraphicsDeviceManager` to App.xaml when the project was created. `SharedGraphicsDeviceManager` creates and manages an instance of the XNA `GraphicsDevice` class.

Whenever an application switches from Silverlight rendering to XNA rendering, it must call the `GraphicsDevice` extension method `SetSharingMode` with a `true` value.

When returning to Silverlight rendering, the same method is called with a `false` value. Normally, `SetSharingMode` is called to enable XNA rendering when navigating to a page:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    SharedGraphicsDeviceManager.Current
        .GraphicsDevice.SetSharingMode(true);
    ...
}
```

Silverlight rendering is restored when navigating away from a page:

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    ...
    SharedGraphicsDeviceManager.Current
        .GraphicsDevice.SetSharingMode(false);
    ...
}
```

The Windows Phone Silverlight and XNA Application project template added the `SetSharingMode` calls to `GamePage` when the project was created. `GamePage` was also created with a `GameTimer`, which is the class that runs the XNA game loop.

14.1.2 The game loop

Silverlight applications are written around events. Work is performed in response to events such as `Click`, `Tap`, or `SelectionChanged`. XNA games run in a polling-based game loop. The game loop starts by checking whether there's work to do. Is the user currently touching the screen? Are there game elements such as enemies or projectiles that need to be moved? Once all the work is finished and all game state has been updated, the game loop draws the relevant graphics to the screen. Finally, the loop stops and waits until the next iteration of the loop should be started. The cycle is repeated—update, draw, update, draw, update, draw—until the game is over. One iteration through the loop is called a *frame*, and the frequency in which the loop executes is called the *frame rate*.

It's the job of the `GameTimer` to execute the game loop. The `GameTimer` instance for the `GamePage` is created and initialized in the page constructor:

```
public GamePage()
{
    ...
    timer = new GameTimer();
    timer.UpdateInterval = TimeSpan.FromTicks(333333);
    timer.Update += OnUpdate;
    timer.Draw += OnDraw;
}
```

The game loop frame rate is specified using the `UpdateInterval` property. The project template generated code that set the `UpdateInterval` property to 333,333 ticks, or 30 frames per second. The `GameTimer` class raises one `Update` and one `Draw` event each iteration of the game loop. These events are wired to the `GamePage` event handler methods `OnUpdate` and `OnDraw`.

The `OnUpdate` and `OnDraw` event handlers are given an instance of `GameTimerEventArgs`. `GameTimerEventArgs` exposes three properties that provide data about the current game state. The `ElapsedTime` property reports the game time elapsed since the last iteration of the game loop. The `TotalTime` property reports the total amount of game time elapsed since the timer started. These two properties report game time, which may not correspond to real time.

Under normal conditions, the game loop calls the `Update` event handler, and then immediately calls the `Draw` event handler. When the game loop is unable to execute the update and draw routines one right after the other, it'll set the `GameTimerEventArgs` property `IsRunningSlowly` to `true`. When the update takes too long, the game loop will skip the call to the draw routine, and will instead call the update routine again. The game loop may skip drawing for several frames while it tries to make up for lost time. In this scenario, `IsRunningSlowly` can be used to help identify bottlenecks in game code.

The `GameTimer` is started in the `OnNavigatedTo` method, and stopped in `OnNavigatedFrom`. The timer's `Start` method will begin running the game loop. The game loop will continue to run until the `Stop` method is called. It's important that the `GameTimer` be stopped before leaving the page or it'll continue to run until the application is exited.

You now have a basic understanding of the game loop and the roles of the update and draw routines. The next step in the game is to implement the update and drawing methods. Before we get to the game loop routines, you need to learn how to use the XNA Framework to build a game.

14.2 Building the game page

In this section we're going to introduce a few XNA topics. Hopefully by the end of the section you'll have a good understanding of the basics of drawing 3D models and how to move a game camera around in a world. In this game, the 3D objects are all drawn from models.

You're going to isolate your XNA code from the `GamePage` code by creating a new class named `GamePlayComponent`. The initial implementation of the `GamePlayComponent` will only have methods that match the game loop routines:

```
public class GamePlayComponent
{
    public void Initialize(ContentManager content) { }
    public void Update() { }
    public void Draw() { }
}
```

Add a field to hold an instance of the `GamePlayComponent` to the `GamePage` class:

```
GamePlayComponent gamePlay = new GamePlayComponent();
```

Before you can implement the `Draw` routine, you must have something to draw. The XNA Framework draws three-dimensional objects using either low-level primitives or prebuilt models created by third-party modeling tools. You're going to use models in this project. What do you need to do to add models to your project?

14.2.1 Understanding models

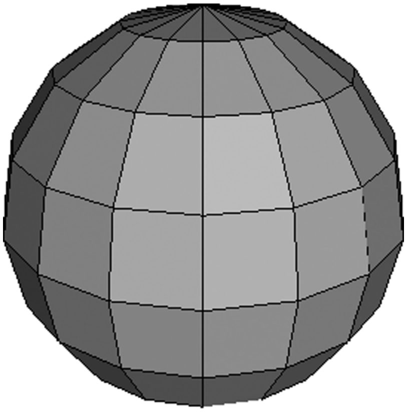


Figure 14.2 A model of a sphere

A *model* is a representation of a physical object built from a collection of points, lines, and faces. The points (vertices) are connected together by lines (edges). Lines are combined to form *faces*, usually in the form of simple geometric shapes such as triangles or quadrilaterals. Figure 14.2 shows how quadrilaterals and triangles are both used to model a sphere. The points, lines, and faces are collectively called a *mesh*. Only the surface of the sphere is described by the model.

Each point on the surface is an (x,y,z) coordinate relative to a nonvisual element named a *bone*. A model may be comprised of several bones, each with its own mesh, and each anchored to a parent bone. The models you use in the sample application are all relatively simple shapes with at most a couple of bones and meshes.

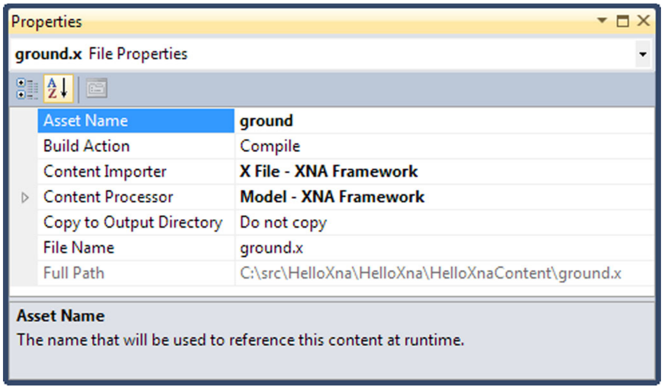
Models are built using third-party 3D design software such as Wings 3D, Blender, or Autodesk Maya. The models you're going to use in your 3D graphics application were created in Wings 3D and will be imported into the project.

> **NOTE** The model files can be found with the book's source code and can be downloaded from the *Windows Phone 7 in Action* page on Manning's website: http://www.manning.com/perga.

Select the content project in the solution explorer and choose *Add Existing Item* from the context or Project menu. Browse to the folder containing the model files and add the cone.x, cube.x, dodecahedron.x, ground.x, octohedron.x, sphere.x, and torus.x files.

The imported models will have their Asset Name property default to the filename. The Asset Name property is the value that will be used when the models are loaded into the game. Figure 14.3 shows the file properties for the newly imported ground.x model.

When the content project is compiled, the model files are processed by the content pipeline. The content pipeline reads the model files and converts them to .xnb files, a format that's efficient to load at runtime by the `ContentManager` class. The `ContentManager` is responsible for loading models, sprite sheets, and other types of game assets that have been compiled in a content project.



Properties

ground.x File Properties

Asset Name	**ground**
Build Action	Compile
Content Importer	**X File - XNA Framework**
Content Processor	**Model - XNA Framework**
Copy to Output Directory	Do not copy
File Name	ground.x
Full Path	C:\src\HelloXna\HelloXna\HelloXnaContent\ground.x

Asset Name
The name that will be used to reference this content at runtime.

Figure 14.3 File properties for the ground.x model

The Windows Phone Silverlight and XNA Application project template you used to create the application adds a `Content` property to the generated `App` class in the App .xaml.cs file. The `Content` property exposes an instance of the `ContentManager` class that you'll use to load model objects. The models will be loaded in an `Initialize` method, shown in the following listing, that you add to the `GamePlayComponent`.

Listing 14.1 Loading models in the `GamePlayComponent` class

```
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

Model ground;
Model player;
Model[] models = new Model[5];

public void Initialize(ContentManager content)
{
    ground = content.Load<Model>("ground");
    player = content.Load<Model>("sphere");
    models[0] = content.Load<Model>("cube");
    models[1] = content.Load<Model>("cone");
    models[2] = content.Load<Model>("dodecahedron");
    models[3] = content.Load<Model>("torus");
    models[4] = content.Load<Model>("octohedron");
}
```

① **Declare member fields**

② **Load models by name**

First you need to add member fields ① to the `GamePlayComponent` class. You need a field for the ground and the player. You store the rest of the models in an array. Next you declare the `Initialize` method with a `ContentManager` parameter. You assign the fields using the `ContentManager`'s `Load` method. The `Load` method accepts a generic type, in this case `Model`, and the name of the asset to load ②.

The `Initialize` method should be called by the `OnNavigatedTo` method in the `GamePage` class:

```
gamePlay.Initialize(contentManager);
```

The `ContentManager` only loads a model file once. Each file is loaded the first time the `Load` method is called, and is managed internally by the `ContentManager` class. When asked for a model a second time, the `ContentManager` returns the instance already loaded in memory.

Now that the models are compiled into your project and loaded by `GamePlayComponent`, you're ready to render the models to the screen.

14.2.2 Rendering models

When working with the XNA Framework, all drawing is explicit—nothing is automatically drawn like in Silverlight. `Model`s are drawn by calling their `Draw` method. Behind the scenes, the `Model` class calls through to the `GraphicsDevice`. Before calling `Model.Draw`, a few steps need to be performed first. The next listing shows the algorithm for drawing a model.

Listing 14.2 Drawing a model

```
using Microsoft.Xna.Framework;

Matrix[] transforms = new Matrix[2];

private void DrawModel(ref Model model, ref Matrix world)
{
    model.CopyAbsoluteBoneTransformsTo(transforms);

    for (int mIndex = 0; mIndex < model.Meshes.Count; mIndex++)
    {
        ModelMesh mesh = model.Meshes[mIndex];
        for (int eIndex = 0; eIndex < mesh.Effects.Count; eIndex++)
        {
            BasicEffect effect = (BasicEffect)mesh.Effects[eIndex];
            effect.EnableDefaultLighting();
            effect.World = transforms[mesh.ParentBone.Index] * world;
            effect.View = cameraView;
            effect.Projection = cameraProjection;
        }
        mesh.Draw();
    }
}
```

❶ **Reusable transforms array**

❷ **Get relative positions**

❸ **Setting up effect**

First you create a reusable `Matrix` array in a class member field named `transforms` ❶ to eliminate the memory allocation and garbage collection that would otherwise result from using a local variable. Inside the `DrawModel` method, you fill the `transforms` array with a call to `CopyAbsoluteBoneTransformsTo` ❷. The `transforms` array now contains information that describes the position of each mesh in the model relative to the position of the model. Next you iterate over each mesh in the model, preparing each `Effect` in the mesh ❸. You tell the effect to use the default lighting algorithm, which creates three lighting sources—a key light, a fill light, and a back light. You set the effect's `World` property by mixing the mesh's position relative to the model with the model's position in the game's coordinate system. Finally you tell the effect where the game camera is located (the `View`) and the boundaries of the field of view (the `Projection`).

In the XNA Framework, `Effects` are used to describe how a `Model` should be rendered by the graphics device. The `Effect` produces High Level Shader Language (HLSL) code, which is interpreted by the GPU, describing exactly how to draw the model. HLSL is a language supported by DirectX GPUs and is beyond the scope of this book. In this project, the model effects are all of type `BasicEffect`.

The `DrawModel` method shown earlier uses two fields named `cameraView` and `cameraProjection`, which you haven't yet defined. The view and projection describe what to draw and from what angle to draw it. The view describes the direction the camera is pointed and is created from a camera position, a look ahead position, and the up direction relative to the camera. The projection describes the portion of the world that the camera can actually see, including the near and far clipping planes. Figure 14.4 demonstrates the view and projection. Only objects in the shaded area of the figure will be drawn.

To calculate the view and projection you need to add a few constants and fields that you'll use in your game logic:

```
Vector3 unitX = Vector3.UnitX;
Vector3 up = Vector3.Up;
readonly Vector3 cameraOffset = new Vector3(0, 5, -15);
readonly Vector3 lookAhead = new Vector3(0, 0, 20);
Matrix cameraView;
Matrix cameraProjection;
```

You calculate the `cameraProjection` in the `Initialize` method. The projection's field of view is set to 45 degrees. The aspect ratio is defined as the view width divided by the view height, and you use the screen width divided by the screen height. You

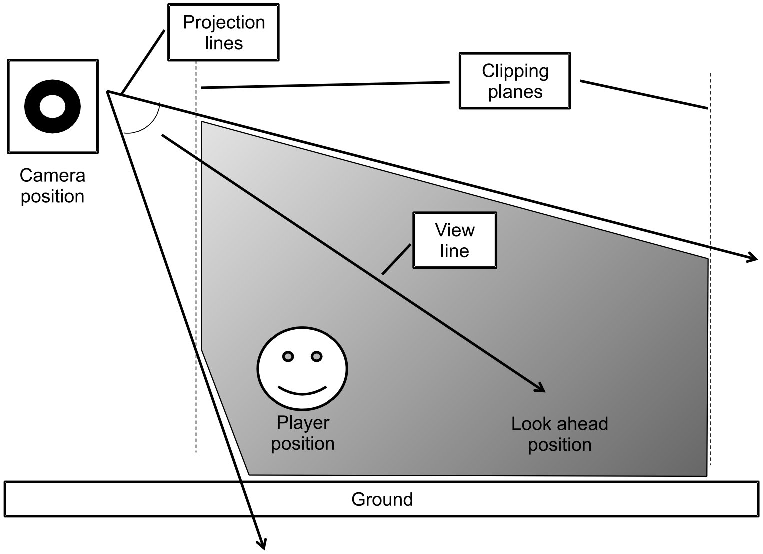


Figure 14.4 The projection created by the camera and look ahead positions, the clipping planes, and the field of view angle

set the near clipping plane to be 1 unit in from of the camera, and the far clipping plane to be 100 units in from of the camera. Add the following line of code to the `Initialize` method:

```
Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
        480.0f/800.0f, 1.0f, 100.0f, out cameraProjection);
```

When drawing models, the camera's position and direction are used as the origin of the coordinate system. The view matrix describes how to transform world coordinates into view coordinates. You calculate the view matrix in a method named `CalculateView`:

```
private void CalculateView()
{
    Vector3 cameraPosition = cameraOffset;
    Vector3 cameraTarget = cameraPosition + lookAhead;

    Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget,
        ref up, out cameraView);
}
```

In this game, the camera will hover above and behind the player's position. Since you haven't implemented player movement yet, the camera is positioned above the world origin by 5 units and behind the origin by 15 units along the Z axis, as defined in the `cameraOffset` field. The look ahead position, or camera target, is placed 20 units in front of the origin along the Z axis, as defined in the `lookAhead` field.

Now that you've established the view and projection, you need to build the world matrix for the ground. The world matrix is the translation of points in the model's coordinate system into the coordinate system of the game:

```
Matrix groundWorld;

private void CalculateWorlds()
{
    groundWorld = Matrix.CreateWorld(Vector3.Zero, Vector3.UnitX,
       Vector3.Up) * Matrix.CreateScale(50.0f, 1.0f, 50.0f);
}
```

The ground model should be drawn at the origin of the world, facing into the positive x direction. The ground model you imported is a square grid 5 units on a side, with the center of the square at the origin. You'd like the ground to cover a larger portion of the world, so you scale the ground's world in the X and Z planes by a factor of 50.

One last step before you draw is to add calls to the calculate methods from the `Initialize` method:

```
public void Initialize(ContentManager content)
{
    ...

    CalculateWorlds();
    CalculateView();
}
```

Now you just need to implement the `Draw` method:

```
public void Draw()
{
    DrawModel(ref ground, ref groundWorld);
}
```

You're now ready to call the draw method from the `OnDraw` method in the `GamePage`:

```
private void OnDraw(object sender, GameTimerEventArgs e)
{
    var device = SharedGraphicsDeviceManager.Current.GraphicsDevice;
    device.Clear(Color.CornflowerBlue);
    device.DepthStencilState = DepthStencilState.Default;
    device.RasterizerState = RasterizerState.CullCounterClockwise;
    gamePlay.Draw();
}
```

When the game runs now, the screen is divided between the newly added ground and the blue sky. Though this 3D world is still not that interesting, at least it's no longer a big blue rectangle. You can make the game slightly more interesting by drawing several shapes.

14.2.3 Adding shapes

The goal of the game is to have the player wander around collecting shapes scattered about the ground. The shapes are going to be drawn in ordered rows, each row containing six shapes of the same model. Before we can draw the shapes, you need to create the world matrix and assign a position for each shape. You need to relate the shape to the model that will be used to draw the shape. To track this shape information, create a new `struct` with the name `ShapeInfo`:

```
struct ShapeInfo
{
    public Matrix World;
    public int ModelIndex;
}
```

The `ShapeInfo` objects are created and initialized in the `CalculateWorlds` method. The following listing shows the new implementation of `CalculateWorlds`.

Listing 14.3 Calculating the position of each shape

```
ShapeInfo[] shapes = new ShapeInfo[66];                  ◁ ❶ Array field to hold shapes

private void CalculateWorlds()
{
    groundWorld = Matrix.CreateWorld(Vector3.Zero, Vector3.UnitX,
       Vector3.Up) * Matrix.CreateScale(50.0f, 1.0f, 50.0f);

    int currentModel = 0;
    int objectIndex = 0;
    Vector3 position = new Vector3();
    for (float x = -100.0f; x <= 100.0f; x += 20.0f)       ❷ Each position in each row
    {
        for (float z = 0.0f; z <= 100.0f; z += 20.0f)
        {
            position.X = x;
            position.Z = z;
            var shape = new ShapeInfo { ModelIndex = currentModel };

            Matrix.CreateWorld(ref position, ref unitX,                ❸ Calculate world
                ref up, out shape.World);                                from position
            shapes[objectIndex] = shape;
            objectIndex++;
        }
        currentModel++;                                                  ❹ Cycle through
        if (currentModel >= models.Length)                                 each model
            currentModel = 0;
    }
}
```

First you create an array field to hold the `ShapeInfo` ❶ objects that will describe each shape. Because you'll have 11 rows with 6 shapes, you size the array to hold 66 shapes. You want to draw a shape every 20 units starting from (-100, 0, 0) and extending to (100, 0, 100). Nested loops are used to iterate over the x and z planes ❷, alternating models during each x loop ❹. At each coordinate, you create a new world matrix ❸ and store it in the ShapeInfo instance.

You're now ready to draw the shapes. Update the `Draw` method by adding a `for` loop to iterate over the shapes array, calling `DrawModel` for each shape. You use the shape's `ModelIndex` property to retrieve the appropriate `Model` from the models array:

```
public void Draw()
{
    DrawModel(ref ground, ref groundWorld);
    for (int index = 0; index < shapes.Length; index++)
    {
        var shape = shapes[index];
        DrawModel(ref models[shape.ModelIndex], ref shape.World);
    }
}
```

When the game runs now, you see several geometric shapes arranged in an orderly grid. The game is improving, but still remains pretty static. You need to allow the player to move around in your geometric world.

14.2.4 Moving around

Two things change when a player moves through a 3D world: the camera position relative to the origin, and the camera's target or look ahead position. Your game currently uses hard-coded camera and look ahead positions. To implement motion, the first thing you need to do is update the code to use member variables for tracking the player position and the direction the player is facing. While you're at it, you're going to add code to draw the player on the screen. To track the direction the player is facing, you record the direction angle relative to the Y axis. You also need a constant for how fast the player turns:

```
Matrix playerWorld;
BoundingSphere playerPosition = new BoundingSphere(
    new Vector3(0,0, -20), 1.0f);
float playerRotation = 0.0f;
const float velocity = 0.5f;
const float rotationVelocity = MathHelper.Pi / 90.0f;
```

Instead of using a single coordinate point to track the player's position, you declare a `BoundingSphere`. The `BoundingSphere` offers several methods that will be useful when you implement collision detection, and we'll take a closer look at `BoundingSphere` and other bounding volumes later in the chapter.

When the player moves, the camera's view must be recalculated. You need to update `CalculateView` because the current implementation doesn't account for the player's position. The new implementation is shown in the next listing.

Listing 14.4 Calculating the camera view to include the player

```
private void CalculateView()
{
    Vector3 cameraPosition = playerPosition.Center +
        Vector3.Transform ( cameraOffset,
            Matrix.CreateFromAxisAngle (Vector3.UnitY, playerRotation));

    Vector3 cameraTarget = cameraPosition + Vector3.Transform(lookAhead,
        Matrix.CreateFromAxisAngle(Vector3.UnitY, playerRotation));

    Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget,
        ref up, out cameraView);
}
```

Rotating offset vector ①

Rotating look ahead vector ②

The original implementation placed the camera 15 units behind the origin, in the direction of the Z axis. You need to place the camera 15 units behind the player in the direction the player is looking and rotate the `cameraOffset` vector using the `Transform` and `CreateFromAxisAngle` ① methods. Next, you add the rotated vector to the player's position. You perform a similar calculation to establish the camera target ②. Finally, you update `cameraView`.

You still haven't implemented any motion, as the `playerPosition` and `playerRotation` values never change. The player will be allowed to move forward and backward, and turn left or right. Ultimately the player's position will change based on input from the touch screen, hardware keyboard, or the motion sensor. Since you don't want to tie your movement update code to a specific input model, you'll create an interface that describes player motions, as shown in the following listing.

Listing 14.5 Input service interface

```
public interface IGamePlayInput
{
    bool MoveForward { get; }
    bool MoveBackward { get; }
    bool TurnLeft { get; }
    bool TurnRight { get; }
```

① **Motion properties**

```
    void Initialize(ContentManager content);
    void Start();
    void Stop();
    void Update();
    void Draw();
}
```

❷ **Game loop methods**

The interface declares a set of motion properties and a set of game loop methods. The motion properties ❶ correspond to the moving and turning motions the player is allowed to make. If the player is currently moving, the motion properties will return `true`. The game loop methods ❷ are provided to allow an input service the opportunity to initialize and clean up before and after the game loop runs. `Initialize`, `Draw`, and `Update` methods are also provided for input components that draw their own user interface.

The `GamePlayComponent` needs to have an instance of the interface passed to it. You'll add a parameter to the `Initialize` method that accepts an input component. The passed-in input component will be stored in a new field:

```
IGamePlayInput input;

public void Initialize(ContentManager content, IGamePlayInput inputService)
{
    input = inputService;
    ...
}
```

Finally you can add an `Update` method to change the player's position and rotation values and recalculate the view. The `Update` method implementation is shown in the next listing.

Listing 14.6 The game's `Update` method

```
public void Update()
{
    bool playerMoved = false;

    Vector3 direction = Vector3.Transform(Vector3.UnitZ,
        Matrix.CreateFromAxisAngle(Vector3.UnitY, playerRotation));

    if (input.MoveForward)
    {
        playerPosition.Center += direction * velocity;
        playerMoved = true;
    }

    if (input.MoveBackward)
    {
        playerPosition.Center -= direction * velocity;
        playerMoved = true;
    }

    if (input.TurnLeft)
    {
        playerRotation += rotationVelocity;
        if (playerRotation > MathHelper.TwoPi)
```

❶ **Did player move?**

❷ **Move player**

❸ **Rotate player**

```
            playerRotation -= MathHelper.TwoPi;
        playerMoved = true;
    }
    if (input.TurnRight)
    {
        playerRotation -= rotationVelocity;
        if (playerRotation < 0)
            playerRotation += MathHelper.TwoPi;
        playerMoved = true;
    }
    if (playerMoved)
    {
        CalculateView();
        Vector3 forward = Vector3.Transform(Vector3.UnitX,
            Matrix.CreateFromAxisAngle(Vector3.UnitY, playerRotation));
        Matrix.CreateWorld(ref playerPosition.Center, ref forward,
            ref up, out playerWorld);
    }
}
```

③ Rotate player

④ Recalculate

First you define a `boolean` variable ❶ to flag when the player moved, which you default to `false`. To move and rotate the player, you use the two velocity constants you defined earlier. The `velocity` value represents how much the player should move per frame, and the `rotationVelocity` represents how much the player should spin per frame. Before you can move the player, you must calculate the direction the player must be moved scaled by the velocity ❷. This direction vector is added to the player's current position. The player's rotation is also updated by the `rotationVelocity` ❸. If the position or rotation values change, you set the `playerMoved` variable to `true`. Finally, you recalculate the camera view and the player's world matrix ❹ only when the player has moved. The camera view is recalculated in the `CalculateView` method. The rotation angle is used to create a forward vector representing the direction the player is facing. The player's world matrix is recalculated using the current position and the forward vector.

Now that you have a moving player, you'll update the `Draw` method to render a sphere at the player's position. Add a new call to `DrawModel` at the bottom of the `Draw` method, using the `player` and `playerWorld` fields:

```
DrawModel(ref player, ref playerWorld);
```

You also need to initialize the `playerWorld` variable in the `CreateWorlds` method, or the player won't be visible until the user starts moving:

```
playerPosition.Radius = player.Meshes[0].BoundingSphere.Radius;
Matrix.CreateWorld(ref playerPosition.Center, ref unitX, ref up,
    out playerWorld);
```

You can't run the game yet because `GamePlayComponent` will throw a `NullReferenceException` when it attempts to use the input service. Before you can run the game, you need to implement an input service.

14.2.5 Running a demonstration

Your first implementation of an input service will run the game with a demonstration script that sends mock movement data to the game. The demonstration mode results in the player moving through a known path in the game and gives you repeatability when building out the rest of the game features. In the next chapter, you'll build input services that accept input from the touch screen and the motion sensor.

Your demonstration script needs to

- Pause for a fraction of a second before motion begins
- Move the player forward for 3 seconds
- Turn the player to the left for 2 seconds
- Turn the player to the right for 4 seconds
- Turn the player back to center
- Move the player backward for 2 seconds
- Move the player forward for 5 seconds
- Turn the player to the left for 3 seconds

You start by creating a new class named `DemoInput` that implements the `IGamePlayInput` interface. `DemoInput` keeps track of the values of the motion properties by using an internal class named `ScriptedInputState`. The following listing details the initial implementation of the `DemoInput` class.

Listing 14.7 The `DemoInput` class

```
public class DemoInput : IGamePlayInput
{
    struct ScriptedInputState
    {
        public bool MoveForward;
        public bool MoveBackward;
        public bool TurnLeft;
        public bool TurnRight;
    }

    ScriptedInputState currentState;

    public bool MoveForward {
        get{ return currentState.MoveForward; }}
    public bool MoveBackward {
        get{ return currentState.MoveBackward; }}
    public bool TurnLeft {
        get{ return currentState.TurnLeft; }}
    public bool TurnRight {
        get{ return currentState.TurnRight; }}

    public void Initialize(ContentManager content) { }
    public void Start() { }
    public void Update() { }
    public void Draw() { }
    public void Stop() { }
}
```

① **Internal struct to report motion**

② **Motion properties return current state**

Unimplemented game loop methods

The `DemoInput` class declares a nested `struct` ❶ named `ScriptedInputState`. The `struct` exposes four `boolean` properties, one for each of the supported motions—forward, backward, turn left, and turn right. `DemoInput` tracks an instance of `ScriptedInputState` named `currentState`. The motion properties defined on the `IGamePlayInput` interface ❷ are implemented using `currentState`.

To update `currentState` you're going to create a script that's a series of `ScriptedInputState` instances. The script defines both the motion of the player and the frame number when that motion should begin. The next listing shows how to add a dictionary mapping a frame number to a `ScriptedInputState` and how to construct and initialize the dictionary in the class constructor.

Listing 14.8 Initializing the scripted input states

```
Dictionary<int, ScriptedInputState> states;

public DemoInput()
{
    states = new Dictionary<int, ScriptedInputState>();
    states.Add(10, new ScriptedInputState { MoveForward = true });
    states.Add(100, new ScriptedInputState { TurnLeft = true });
    states.Add(160, new ScriptedInputState { TurnRight = true });
    states.Add(280, new ScriptedInputState { TurnLeft = true });
    states.Add(340, new ScriptedInputState { MoveBackward = true });
    states.Add(430, new ScriptedInputState { MoveForward = true });
    states.Add(750, new ScriptedInputState { TurnLeft = true });
    states.Add(840, new ScriptedInputState());
}
```

Field to store script

❶ **Move forward**

❷ **Stop moving**

After the `states` dictionary is created, you add several `ScriptedInputState` instances. The first instance ❶ tells the game that at frame number 10, the player starts moving forward. The player will move forward until frame 100, at which point the player will stop moving forward and will begin turning left. The player moves forward for 90 frames, or 3 seconds when the game runs at 30 frames per second. Motion continues until frame number 840 ❷ when the player stops moving.

Next you implement the input service's `Update` method. Every time the `Update` method is called, you increment a frame counter field. You use the frame counter to look for a new input state in the dictionary, as implemented in the next listing.

Listing 14.9 Updating current state

```
int frameNumber;

public void Update()
{
    frameNumber++;

    ScriptedInputState nextState;
    if (states.TryGetValue(frameNumber, out nextState))
    {
```

Increment frame number

```
            currentState = nextState;
        }
    }
```

① **Update currentState**

The `frameNumber` field is used as a key into the `states` dictionary. When a `ScriptedInputState` is found that's keyed with the current frame number, the `currentState` field ① is updated.

This completes the `DemoInput` service and it's now ready to be used by the `GamePage`. Add an `IGamePlayInput` field to the `GamePage` class and initialize the field in the `OnNavigatedTo` method. Pass the new `input` instance to the `gamePlay` component:

```
input = new DemoInput();
gamePlay.Initialize(content, input);
```

The `DemoInput` class doesn't work unless its `Update` method is called by the game loop. Add the call to the `Update` method from `GamePage`'s `OnUpdate` method:

```
private void OnUpdate(object sender, GameTimerEventArgs e)
{
    input.Update();
    gamePlay.Update();
}
```

Finally you have a game that draws a three-dimensional world and moves the player. If you run the game now, you might notice a strange behavior. When the player's position intersects a shape's position, the player passes right through. You'll fix this quirk later in the chapter when we look at collision detection. First we need to discuss a trick to help preserve the phone's battery life.

14.2.6 Don't repeat yourself

Developers of Windows Phone applications must keep the health of the phone's battery in mind when building applications. This applies when using the radio for network connections and also applies when updating the screen. Every time the display is redrawn, a bit more power is consumed. You can help extend the battery life by only clearing and redrawing the screen when absolutely necessary.

You can tell the game loop to not redraw by using the `GameTimer`'s `SuppressFrame` method. When `SuppressFrame` is called, the game loop won't make any calls to the `Draw` method until after the next call to `Update`.

In this game, you should skip the `Draw` method whenever the player isn't moving. You can detect when the player is moving using the input component. The `OnUpdate` method is where the motion should be examined. If the player didn't move, there's no need to redraw the screen and you should call `SuppressFrame`. The next listing details the new `OnUpdate` implementation that suppresses unnecessary redraw operations.

Listing 14.10 Suppressing redraw

```
int frameCountAfterMotionStopped;

private void OnUpdate(object sender, GameTimerEventArgs e)
{
    input.Update();
    gamePlay.Update();

    if (!input.MoveForward && !input.MoveBackward
        && !input.TurnLeft && !input.TurnRight)
    {
        if (frameCountAfterMotionStopped > 2)
        {
            GameTimer.SuppressFrame();
        }
    }
    else
    {
        frameCountAfterMotionStopped = 0;
    }
}
```

New field

❶ **Did player move?**

❷ **Draw couple extra frames**

❸ **Suppress redraw**

First you check each of the motion properties of the input service ❶. If none of them are set to `true`, you know the player hasn't moved. You can't just suppress the `Draw` operation if the player hasn't moved. You need to be sure to draw the screen at least once. Consider the scenario when the game is first started: if you don't draw at least once, the screen will remain blank until the player starts to move. Before you suppress the next `Draw` call, you check whether the screen has been drawn a couple times after motion stopped ❷ using the new field `frameCountAfterMotionStopped`. If the screen has been drawn you suppress the next redraw ❸ by calling `SuppressFrame`.

You draw a couple extra frames to allow the screen to be drawn when the game starts and the play has never moved. There may also be situations when an input component needs a couple of extra frames to fully update the screen.

If you did your work correctly, you shouldn't notice any differences when you run the game. The demonstration script still moves the user around the world. You still have that pesky problem where the player walks through the shapes. Instead of walking through them, the shapes should disappear when the player touches, or collects, them.

14.2.7 Collecting shapes

One of the goals of the game is to collect the shapes and rack up some points for each shape collected. Right now the game completely ignores the shapes and the player walks right through them without detecting their presence. What you need is some code that tracks the score and allows you to determine when the player's position intersects with a shapes position. You'll start by adding a `Score` property that's increased by one every time the player collides with a shape, and the shape is collected:

```
public int Score { get; private set; }
```

One method for detecting collisions is to check the (x,y,z) coordinate of the player with the coordinate of the shape. But shapes aren't points; they have volume and if you only check the position coordinate, the user may notice that they collided with a shape, but the collision didn't register.

The XNA Framework provides several structs that help detect collisions. You'll be using the `BoundingSphere` struct. We introduced the `BoundingSphere` when you added the player position earlier in the chapter. The XNA Model creates `BoundingSpheres` for the meshes that make up the model. `BoundingSpheres` are simple to work with since they only have a center point and a radius.

> **TIP** The `BoundingSphere` class is one of the XNA Framework structures useful for 3D coordinate system calculations. These structures can be used by your application, even if you're not using XNA Graphics.

You'll capture the bounding spheres of each model and store it along with the rest of the shape information by adding a `BoundingSphere` field to your `ShapeInfo` struct:

```
struct ShapeInfo
{
    public Matrix World;
    public int ModelIndex;
    public BoundingSphere Sphere;
    public bool Collected;
}
```

While you're there, go ahead and add a Boolean field named `Collected` to the struct. You'll use the `Collected` field as a flag to determine when a shape has been touched or picked up by the player and should no longer be considered when performing collision detection and drawing.

Next you need to update the code that initializes the `ShapeInfo` instances in `CalculateWorlds`. When building shape information, you need to set the sphere's position and copy the radius from the model's `BoundingSphere`:

```
var shape = new ShapeInfo { ModelIndex = currentModel };
shape.Sphere.Center.X = x;
shape.Sphere.Center.Z = z;
shape.Sphere.Center.Y = models[currentModel].
                            Meshes[0].BoundingSphere.Center.Y;
shape.Sphere.Radius = models[currentModel].Meshes[0].BoundingSphere.Radius;
```

In this project, all of the models are simple and using the `BoundingSphere` from the first mesh is sufficient. In models with multiple bones and meshes, you should customize your collision detection logic to account for the additional complexity. The collision detection logic shown in the following listing is placed in the `GamePlayComponent.Update`, inside the `if` block that executes when the player moves.

Listing 14.11 Detecting collisions in the `Update` method

```
using Microsoft.Devices;

if (playerMoved)
{
    ...
    for (int index = 0; index < shapes.Length; index++)
    {
        if (!shapes[index].Collected &&
            playerPosition.Intersects(shapes[index].Sphere))
        {
            Score++;
            shapes[index].Collected = true;
            VibrateController.Default.Start(
                TimeSpan.FromMilliseconds(20.0));
        }
    }
}
```

❶ Collision when the spheres intersect

❷ Update score

❸ Briefly vibrate phone

You start by iterating over the shapes array, ignoring the shapes that have already been collected. You use `BoundingSphere`'s `Intersects` ❶ method to determine whether the shape and the player are touching. If the two `BoundingSpheres` do intersect, the `Score` property is increased ❷. Finally, you vibrate the phone for 20 milliseconds ❸ using the `VibrationController` class.

> **TIP** The `VibrateController` can be used in any type of application or game, not just those mixing Silverlight and XNA graphics. Vibrating the device consumes extra battery power and may be annoying to the user. Consider providing an option allowing the user to turn off vibrations.

The `VibrationController` is found in the `Microsoft.Devices` namespace. The `VibrationController` is a singleton accessed through the static method `Default`. Vibrations are started with the `Start` method. A `TimeSpan` is used to specify how long to vibrate the phone and valid `TimeSpan` values are between 0 and 5 seconds.

Once a shape is collected, it disappears from the screen. To perform the disappearing trick, you simply no longer draw the shape. Add an `if` statement to the `Draw` method so that you only draw a shape if its `Collected` property is `false`:

```
if(!shape.Collected)
    DrawModel(ref models[shape.ModelIndex], ref shape.World);
```

Finally you have a game that draws a three-dimensional world, allows the player to move about and explore, and detects collisions between the player and the other shapes. The last thing you need to add in this chapter is how to recognize when the game ends.

14.2.8 It's the end of the world

Your 3D graphics application isn't complete until the game is over, and the game you've built doesn't end. You have a world with a ground of limited size, but the player can wander far outside the edge of that world. The player could also collect all the shapes and have nothing else to do. You need to add some logic to detect when all the shapes are collected, or the player has moved past the edge of the world, and end the game. When the game is over, your application should automatically navigate back to MainPage.

You begin your game-over implementation by adding a property to the `GamePlayComponent` to distinguish between when the game is in play and when the game is over. The `IsPlaying` property should be initialized to `true` in the `Initialize` method:

```
public bool IsPlaying { get; private set; }
```

Next, you need a technique to determine when the user has wandered past the edge of the world, or more precisely, past the bounds of the ground model. A flat square isn't well represented by a `BoundingSphere`, so you need to use another bounding geometry. The `BoundingBox` struct best fits your requirements. Add a new field named `groundBounds` that matches the size of the ground model after you scale it up when it's drawn:

```
BoundingBox groundBounds = new BoundingBox(
    new Vector3(-125, -2, -125), new Vector3(125, 2, 125));
```

Now that you've established the edges of the world, you can check whether the player moved outside the bounds of the world. In addition to the bounds check, you also need to check whether all the shapes have been collected. Your game-over logic, shown in the following listing, is placed in the `GamePlayComponent.Update`, inside the `if` block that executes when the player moves.

Listing 14.12 Detecting game over in the `Update` method

```
if (playerMoved)
{
    ...
    var containment = groundBounds.Contains(playerPosition);
    if (containment == ContainmentType.Disjoint
            || Score == shapes.Length)
    {
        IsPlaying = false;
    }
}
```

➊ **Is player over edge?**

➋ **Have all shapes been collected?**

You check whether the player walked over the edge of the world using `BoundingBox`'s `Contains` method ➊. The `Contains` method returns an instance of `ContainmentType`, which will have one of three values. Possible `ContainmentType` values are `Contains`, `Intersects`, or `Disjoint`. `Contains` indicates that the `BoundingBox` completely contains the player's `BoundingSphere`, whereas `Intersects` signifies only `partial` containment.

You use `Disjoint` in your game-over check, which indicates that the player is completely outside of the `BoundingBox`. You also compare the `Score` property to the length of the shapes array ❷. If either game-over condition is `true`, you set the `IsPlaying` property to `false`.

When the game is over, you no longer process player input and no longer need to execute the code in the `Update` method. Add logic to the top of the `Update` method to return immediately if the game is over:

```
public void Update()
{
    if (!IsPlaying)
        return;
    ...
}
```

You also have nothing to draw when the game is over. Add logic to the top of the Draw method to return immediately if the game is over:

```
public void Draw()
{
    if (!IsPlaying)
        return;
    ...
}
```

You also want to check `IsPlaying` at the end of the `OnUpdate` method in `GamePage` and navigate to the previous page when the game is over:

```
private void OnUpdate(object sender, GameTimerEventArgs e)
{
    ...
    if (!gamePlay.IsPlaying && NavigationService.CanGoBack)
        NavigationService.GoBack();
}
```

The user can now relax knowing that they won't be trapped in the game forever. The game will end if they walk over the edge of the world, or pick up all the shapes.

How does the user know how many shapes they have collected? The game tracks the number of shapes collected in the `Score` property. You need a scoreboard to display the player's progress.

14.3 Implementing a scoreboard with Silverlight

Almost all games require some form of scoreboard. If the player can't track their progress, what's the point of playing? To demonstrate mixing XNA rendering with Silverlight components, your scoreboard will be built using a `Border` containing `StackPanel` and a couple of `TextBlock` controls. The scoreboard will be defined in GamePage.xaml like any other Silverlight page. Because GamePage.xaml is rendered with an XNA `GraphicsDevice`, your scoreboard will be rendered with a `UIElementRenderer`.

In a regular XNA project, the scoreboard would be drawn using the XNA Framework's `SpriteFont` and `DrawString` APIs. A `SpriteFont` is a sprite sheet or image built by the content pipeline from a font installed on a developer's machine. The compiled font is then included in your project .xap file. You have to redistribute each font used in your game. If you use the font in multiple point sizes, you must include a `SpriteFont` for each size. You must include a `SpriteFont` for each language you support as well. You must also ensure that you have an appropriate redistribution license for each font you ship in your application. You can see how using `SpriteFonts` quickly becomes a problem.

By mixing Silverlight controls with XNA rendered graphics, the task of managing fonts disappears. Silverlight ships with a variety of fonts and languages that you don't have to include in your application's .xap file. You'll leverage these built-in fonts with your scoreboard implementation.

14.3.1 Adding a scoreboard

Before you can add components to GamePage.xaml, you need to define a root layout to hold your scoreboard components. GamePage.xaml was generated by the project template as an empty page. Open GamePage.xaml and replace the *No XAML Content* comment with the XAML markup shown in the next listing.

Listing 14.13 XAML markup for the scoreboard

```
<Canvas x:Name="LayoutRoot">                                   ❶ Root layout container
    <Border x:Name="scoreboard"  Canvas.Left="12" Canvas.Top="12"
        Width="456" Height="60" BorderThickness="5" BorderBrush="White">
        <Border.Background>
            <LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">
                <GradientStop Color="SaddleBrown" Offset="0.488" />
                <GradientStop Color="BurlyWood" Offset="1" />
            </LinearGradientBrush>
        </Border.Background>                                   Scoreboard ❷
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Score:"
                Style="{StaticResource PhoneTextLargeStyle}" />
            <TextBlock x:Name="score"
                Style="{StaticResource PhoneTextLargeStyle}" />
        </StackPanel>
    </Border>                                  Display Score with TextBlock ❸
</Canvas>
```

First, you add a `Canvas` container ❶ and give it the conventional name `LayoutRoot`. Next you declare the `Border` control ❷ named `scoreboard`, explicitly setting the position, width, and height properties. You also set the border width and color. Next, you tell the `Border` to use a gradient brush when drawing its `Background`. Finally, you create two `TextBlocks`. The first `TextBlock` is a label displaying the word Score. The second `TextBlock` displays the actual score value ❸, and you name it `score` so that you can access the control from code behind.

You update the score `TextBlock` from code behind in the `OnUpdate` method:

```
private void OnUpdate(object sender, GameTimerEventArgs e)
{
        input.Update();
        gamePlay.Update();
        score.Text = gamePlay.Score.ToString();
        ...
    }
```

If you run the project now, you still won't see a scoreboard. Just because there are XAML elements on the page doesn't mean they're automatically rendered. Remember that you turned off Silverlight rendering when you called `SetSharingMode(true)`. So how do you render the Silverlight components?

14.3.2 Rendering the texture

When in sharing mode, Silverlight components are rendered in a two-step process. The first step renders the user interface to an XNA `Texture2D`. The second step renders the texture with a `SpriteBatch`, which we'll examine next. `Texture2D` is the class XNA uses to represent a two-dimensional image comprised of pixels. `Texture2D` is used for images of different formats and color depths. When you add a .PNG file to a content project, it'll be loaded with a `Texture2D`. You'll learn how to import and use your own image in the next chapter when you work with sprite sheets.

Silverlight components don't know anything about XNA or how to render themselves to an XNA `Texture2D` image. A new class has been added to the XNA Framework that does know how to render Silverlight to a `Texture2D`. This new class is named `UIElementRenderer` and it's found in the `Microsoft.Xna.Framework.Graphics` namespace alongside `Texture2D`, `Model`, `SpriteBatch`, and other XNA graphics classes.

`UIElementRenderer` has two properties, `Element` and `Texture`, and one method named `Render`. The `Element` property references a Silverlight `UIElement`, which could be a single control, a container holding several controls, a shape, an entire page, or any other class derived from `UIElement`. The `Texture` property references the `Texture2D` that can be used by XNA's `SpriteBatch` to draw the user interface. The texture is created by the `Render` method, and isn't automatically updated.

You need to add a `UIElementRenderer` to draw the scoreboard. Add two new fields to the `GamePage` class for the renderer and the screen coordinates where the scoreboard will be drawn:

```
UIElementRenderer scoreboardRenderer;
Vector2 scoreboardPosition = new Vector2(12, 12);
```

The renderer should be constructed at the end of the `GamePage` constructor. You construct the renderer using the `Border` control you named scoreboard. You also tell the render to produce a texture with the same width and height you gave the scoreboard in the XAML markup:

```
scoreboardRenderer = new UIElementRenderer(scoreboard, 456, 60);
```

Now that you have a renderer for your scoreboard, let's look at how to use it to draw the Silverlight user interface. The `GamePage` is drawn in the `OnDraw` method, which is where you add a call to `Render`. Add the call to `Render` to the top of the method, before the code that works with the `GraphicsDevice`. If you call `Render` after you begin drawing, you might end up with elements missing from the screen as the `UIElementRenderer` clears the back buffer while it works:

```
private void OnDraw(object sender, GameTimerEventArgs e)
{
    scoreboardRenderer.Render();
    var device = SharedGraphicsDeviceManager.Current.GraphicsDevice;
...
```

Once the texture is rendered, you use a `SpriteBatch` to draw. Place the `SpriteBatch` code at the end of the `OnDraw` method:

```
spriteBatch.Begin();
spriteBatch.Draw(scoreboardRenderer.Texture,
    scoreboardPosition, Color.White);
spriteBatch.End();
```

Sprite is another term for a texture or an image. The `SpriteBatch` class allows several sprites' drawing operations to be batched together and sent to the `GraphicsDevice` as a single job. A batch operation is started with a call to `SpriteBatch.Begin`. After rendering the scoreboard, you ask your `SpriteBatch` instance to draw the scoreboard at the coordinate described in the `scoreboardPosition` field. When you call `SpriteBatch.End`, the texture is sent to the `GraphicsDevice`.

Run the application now, and you should see the scoreboard at the top of the screen. As the player is moved around with the demonstration script, you should see the score increase when shapes are collected. In the next chapter, you'll add new `IGamePlayInput` implementations that allow the player to be controlled by the user instead of by a script.

14.4 Summary

Though we're finished with this chapter, we're not finished with the game. We'll continue working on the game in the next chapter. In this chapter we showed you how to build a game that mixes Silverlight and XNA programming. Along the way, we gave you a crash course in XNA Framework concepts such as the game loop, cameras, three-dimensional models, and two-dimensional textures. We also showed you how to use bounding geometries to perform collision detection. Finally, you learned how to render Silverlight controls using `UIElementRenderer` and `SpriteBatch`.

The XNA Framework was built as a gaming platform, and your sample project is a game. Games don't have exclusive rights to three-dimensional modeling and rendering. There are a number of potential uses of 3D technology including data visualization, simulations of real-world environments, and representing connections between systems of related objects.

In the next chapter you're going to learn how to receive input from a player using Silverlight's `Button`, XNA's `TouchPanel`, and the `MotionSensor`. We'll show you how to implement a pause-and-resume feature. You're also going to replace your demo script with a variety of input components that use the touch panel and the motion sensor. You'll replicate the thumbstick and button pad controls you'd find on an Xbox 360 controller. You're also going to create components that will control the game with touch gestures or the motion sensor.

15

XNA input handling

This chapter covers

- Pausing a game
- Handling Silverlight events
- Working with touches
- Using the motion sensor

The XNA Framework provides for a number of input devices across all of its target platforms. Game players on an Xbox use the controller with its variety of buttons, triggers, and thumbsticks, as well as a directional pad. Players of Windows-based games can use the controller as well as the keyboard and mouse. Zune devices provide a more limited set of buttons, directional pad, and thumbstick, whereas the Zune HD provides a touch screen and accelerometer.

Applications and games running on the Windows Phone receive all of their player input via the touch screen or sensors. Though the phone has a number of hardware buttons, applications and games can't use any of them for game play. The only exception is the Back button, which games can use to navigate between screens and to exit the application. Some phones have physical keyboards that can be used by application code, but applications and games must also work on phones that don't have physical keyboards. The software keyboard can't be used by the

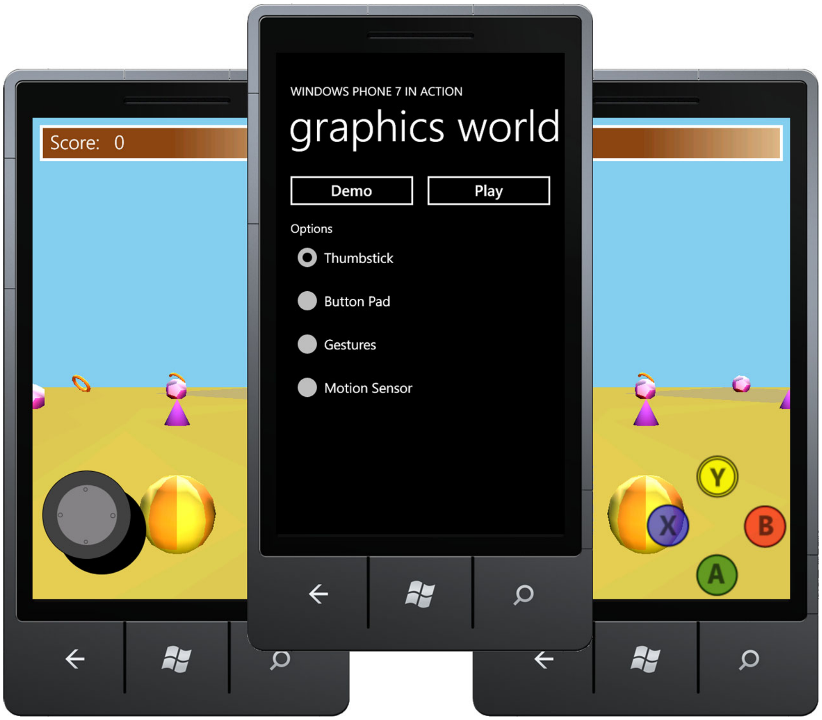
XNA Framework, but can be exposed by Silverlight text input controls even in shared graphics mode.

Touch input is processed by the XNA Framework and surfaced to the developer through a few different APIs. Raw multi-touch data is accessible as `TouchLocations`. For simpler scenarios, touches are transformed and exposed by the Mouse API. Touches are also processed, combined, manipulated, and served up as `GestureSamples` by the Gesture API.

Silverlight elements rendered by the XNA Framework can still use traditional controls to receive input. You start this chapter by adding a Silverlight `Button` control to the game you started in the last chapter, allowing the player to resume game play when the application has been interrupted by the press of the Back button.

Throughout the rest of the chapter, we're going to explore how to use each of the touch-related APIs in a game. You'll also learn how to incorporate the motion sensor as an alternative to touch input. To show off these topics, you'll extend the application from the last chapter to allow the player to choose which input mechanism they want to play with. The revised game is shown in figure 15.1.

The MainPage will provide a radio button for each of the available input types. When the user presses the Play button, the input type will be read from the radio button controls and passed to GamePage.xaml as a query string parameter. The Demo



WINDOWS PHONE 7 IN ACTION

graphics world

Options

● Thumbstick

○ Button Pad

○ Gestures

○ Motion Sensor

Figure 15.1 The XNA Input sample game showing start screen and game play screens with thumbstick and buttons

button will launch `GamePage` with the demonstration script you used in the last chapter. You'll then create four different input mechanisms, each using a different API. The first input service will be a virtual thumbstick implemented with the `TouchPanel.GetState` API. The second input service will use the Mouse API to implement a virtual button pad, another reproduction of the Xbox controller. The last two input services will be implemented with gestures and the motion sensor.

Before you implement the input services, you should learn how to use a Silverlight `Button` control on the XNA-rendered GamePage.

15.1 Implementing pause and resume

The *Application Certification Requirements for Windows Phone* state that games should be paused with the hardware Back button. When a game is active and the user presses the Back button, the game should pause. If the Back button is pressed a second time, the game should navigate to the previous page.

What does it mean for a game to be paused? The game should no longer update—players stop moving, enemies stop chasing, projectiles freeze in place, game countdown timers stop counting down, and the score doesn't change. Input is no longer processed and touching the screen no longer has any impact, except to resume the game. A paused game should provide some visual indication as well as a button or other control to resume the game.

The XNA Framework doesn't include any common user interface controls such as a button. If you were to implement a button using just the XNA Framework, you'd need to detect touches to the screen, check whether a touch was over the button, and animate the button to provide feedback. Fortunately, you don't need to create an XNA button control as you can use Silverlight's button control in your game. Before you add the Resume button, you need to add the ability to pause the game.

15.1.1 Pausing game play

When the game is paused, you need to cease all update operations such as retrieving user input and moving the player. Update operations can be skipped by not calling the input service and game component `Update` methods within the `GamePage.OnUpdate` method. This simplistic approach won't necessarily work in a more complex game, especially one that uses the timing values passed in the `OnDraw` and `OnUpdate` event arguments.

Add a new field to the `GamePage` class and name the field `isPaused`. This field will be used throughout the `GamePage` class to identify when the game is paused:

```
bool isPaused;
```

Most of the game logic that's relevant to a running game is called from the `OnUpdate` method. If you short-circuit the `OnUpdate` then you effectively stop the game in its tracks. Add a check at the top of the `OnUpdate` method that returns immediately when `isPaused` is true:

```
private void OnUpdate(object sender, GameTimerEventArgs e)
{
    if (isPaused)
        return;
    ...
}
```

The game will be paused when the user presses the hardware Back key. In chapter 2 we showed you how a Silverlight application detects a Back button press by overriding the `OnBackKeyPress` method in the page class. The following listing details the `OnBackKeyPress` implementation that pauses your game.

Listing 15.1 Overriding `OnBackKeyPress`

```
protected override void OnBackKeyPress(CancelEventArgs e)
{
    if (!isPaused)
    {
        isPaused = true;
        resumeButton.IsEnabled = true;
        e.Cancel = true;
    }
    base.OnBackKeyPress(e);
}
```

❶ **Do nothing if already paused**

❷ **Cancel navigation**

You start by checking the value of `isPaused` ❶. If `isPaused` is already `true`, you do nothing and let the framework proceed as normal, performing a `GoBack` navigation to `MainPage`. If `isPaused` is `false`, you change its value to `true`, enable the `resumeButton`, and cancel the page navigation ❷ by setting the event argument's `Cancel` property to `true`. Note that you're referencing a `resumeButton` field that you haven't added yet.

> **NOTE** The XNA Framework provides the GamePad API, which can also be used to detect back button presses. Using the GamePad API is beyond the scope of the book.

That's all there is to pausing the game; we told you it was simple. Unfortunately, the game won't compile at this point because of the missing `resumeButton` field. You need to add the Resume button and restore the game to a running mode when it's clicked.

15.1.2 Adding the resume button

To get your application to compile and run, you need to add a Silverlight `Button` control. Just like you did in the last chapter, you declare Silverlight controls in GamePage .xaml. You add the `Button` to the `Canvas` that's your layout root. Make sure to name the button `resumeButton`. The button's content is set to *Resume Game*, and a click event handler is declared:

```
<Canvas x:Name="LayoutRoot">
    ...
    <Button x:Name="resumeButton" Canvas.Left="115" Canvas.Top="364"
        Width="250" Height="72" IsEnabled="False"
        Content="Resume Game" Click="resume_Click" />
</Canvas>
```

The button is initially disabled. This is required because even when the button isn't drawn and is invisible, a touch to the location where the button exists will trigger a button click. The button is positioned at coordinate (115, 364), with a width of 250 and a height of 72. You need these details to define a `UIElementRenderer` and a `buttonPosition` field in the `GamePage` class. The `UIElementRenderer`, which you'll name `buttonRenderer`, is used to render and draw the button control:

```
UIElementRenderer buttonRenderer;
Vector2 buttonPosition = new Vector2(115, 364);
```

The button renderer is constructed in the `GamePage` constructor using the same width and height you specified in the XAML markup:

```
buttonRenderer = new UIElementRenderer(resumeButton, 250, 72);
```

Now you're ready to use the button renderer to draw the button. You're going to update the `OnDraw` method to render and draw the button when the game is paused. You add the call to `Render` to the top of the `OnDraw` method, before you start drawing:

```
private void OnDraw(object sender, GameTimerEventArgs e)
{
    scoreboardRenderer.Render();
    if (isPaused)
        buttonRenderer.Render();
    var device = SharedGraphicsDeviceManager.Current.GraphicsDevice;
...
```

Then you add code to draw the button between the calls to `SpriteBatch`'s `Begin` and `End` methods:

```
spriteBatch.Begin();
spriteBatch.Draw(scoreboardRenderer.Texture, scoreboardPosition,
   Color.White);
if (isPaused)
    spriteBatch.Draw(buttonRenderer.Texture, buttonPosition, Color.White);
spriteBatch.End();
```

The button is drawn by the page's `SpriteBatch` instance, so the new code is added before the call to the `End` method. You put the call to `Draw` inside an `if` block to avoid the extra work when it's not required.

The last task in your pause-and-resume feature is to actually resume the game. You paused the game by setting `isPaused` to `true`, and can resume the game by setting `isPaused` to `false`. When the game is resumed, you also need to disable the button again so that it doesn't trigger any additional button clicks while it's hidden:

```
private void resume_Click(object sender, RoutedEventArgs e)
{
    resumeButton.IsEnabled = false;
    isPaused = false;
}
```

Now you have a game that can be paused and resumed. Still, it's a boring game since the user can't control it—they're just a spectator watching the demonstration script control the player. You're now ready to implement a few different input services, giving the user a choice of how to control the player's movement.

15.2 Adding input services

At this point your game implementation doesn't allow the user to control the game. In this section, you're going to extend the game so that you can switch between the Touch API, the Mouse API, the motion sensor, and gestures. Let's review a couple of key points in the `GamePlayComponent`. The `playerPosition` and `playerRotation` fields track the location of the player and the direction the player is facing. These fields are then used to locate the camera, which in turn is used to draw the models in the world. The `playerPosition` and `playerRotation` are changed by reading values from the `IGamePlayInput` interface. You created the `IGamePlayInput` interface to abstract player movement from the API used to gather input.

Even though the primary input hardware on the Windows Phone is the touch panel, the XNA Framework wraps touch input behind a few different APIs. We're going to look at how `TouchLocations` are provided by the raw touch API, which you'll use to create a thumbstick. The other APIs that make use of touch input are `Mouse.GetState` and `TouchPanel.ReadGesture`. One alternative to touch-based input is the motion sensor.

> **NOTE** The `Keyboard.GetState` API can be used on devices that contain a hardware keypad. Games that use the keypad must also provide an alternate input mechanism for devices that don't have a keypad. We don't cover the keyboard in this book.

In this section you'll create four input services. You'll use the `Mouse.GetState` API to implement a button pad. You'll also create an input service that translates drag gestures into player movement. Finally, you'll design an input service that uses the motion sensor.

Before you implement the new input services, you need to update your user interface to allow the user the ability to choose which input service to use.

15.2.1 Choosing an input type

This game will use radio buttons on the main page to allow the player to choose the input service. The game will also let the user run the game with the demonstration script you built in the last chapter. Figure 15.1 shows the new main page that includes Demo and Play buttons along with four radio buttons. The new `ContentPanel` markup is shown in the next listing.

Listing 15.2 Redesigning the main page

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
```

① Two rows and two columns

```
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Button Content="Demo" Click="demoGame" />
    <Button Content="Play" Grid.Column="1" Click="Button_Click" />

    <StackPanel Grid.Row="1" Grid.ColumnSpan="2" Margin="12">
        <TextBlock Text="Options" />
        <RadioButton x:Name="thumbstick" Content="Thumbstick"
            IsChecked="True" />
        <RadioButton x:Name="buttonPad" Content="Button Pad" />
        <RadioButton x:Name="gestures" Content="Gestures" />
        <RadioButton x:Name="motionSensor" Content="Motion Sensor" />
    </StackPanel>
</Grid>
```

① **Two rows and two columns**

② **Two Buttons**

③ **Four RadioButtons**

First, you start by dividing the `ContentPanel` grid into two rows and two columns ①. The first row will contain two buttons ②. The first button is new, is used to launch the demo, and is wired to a `Click` event handler named `demoGame`. The second button is used to play the game and was already defined as MainPage.xaml. You change the Play button markup so that the button appears in the first row and second column of the `ContentPanel`. The second row in the `ContentPanel` contains a `StackPanel`, a label, and four `RadioButtons` ③. There's one `RadioButton` for each of the input services you're going to implement in this chapter.

When the Play button is clicked, the `Button_Click` event handler is called. Inside the event handler, you examine the `RadioButtons` and add a query string parameter to the `Url` used to navigate to the game page. The new implementation of `Button_Click` is shown in the following listing.

Listing 15.3 Reimplementing the `playGame` event handler

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    int input = 0;
    if (thumbstick.IsChecked.HasValue && thumbstick.IsChecked.Value)
        input = 1;
    else if (buttonPad.IsChecked.HasValue && buttonPad.IsChecked.Value)
        input = 2;
    else if (gestures.IsChecked.HasValue && gestures.IsChecked.Value)
        input = 3;
    else
        input = 4;
    NavigationService.Navigate(GamePage.BuildNavigationUri(input));
}
```

Determine chosen option ①

Add query string parameter ②

The `Button_Click` method was originally generated by the project template when you first created the project in the last chapter. You update the method by adding several `if` statements ① that check the `IsChecked` properties of the `RadioButtons`. When you find the `RadioButton` that's checked, you assign a corresponding value to the `input` variable. You add the value of the variable to a query string parameter ❷ generated by a new GamePage method named `BuildNavigationUri`. Finally, you ask the `NavigationService` to navigate to GamePage.xaml.

Add the `BuildNavigationUri` method shown next to GamePage.xaml.cs:

```
public static Uri BuildNavigationUri(int input)
{
    return new Uri("/GamePage.xaml?Input=" + input, UriKind.Relative);
}
```

Returning to MainPage.xaml.cs, you use a similar query string in the handler for the demo button's `Click` event, named demoGame. In the demoGame method, you set the query string parameter to a value of zero, and navigate to GamePage.xaml:

```
private void demoGame(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(GamePage.BuildNavigationUri(0));
}
```

Now that you're sending the user's choice to the `GamePage`, you need to update `GamePage`'s code behind to look at the query string and react appropriately. The next listing shows the updated `OnNavigatedTo` method in GamePage.xaml.cs that examines the input type.

Listing 15.4 Constructing the input component

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    SharedGraphicsDeviceManager.Current.GraphicsDevice
        .SetSharingMode(true);
    spriteBatch = new SpriteBatch(
        SharedGraphicsDeviceManager.Current.GraphicsDevice);

    string inputType = this.NavigationContext.QueryString["Input"];
    switch (inputType)
    {
        default:
            input = new DemoInput();
            break;
    }
    input.Initialize(content);
    gamePlay.Initialize(content, input);

    timer.Start();
    input.Start();
    base.OnNavigatedTo(e);
}
```

❶ Branch based on input value

❷ Run demo script by default

❸ Load input service content

❹ Start input processing

You start by adding code to extract the input value from the query string, assigning it to the `inputType` variable. Next you add a `switch` statement ❶ that will ultimately create all the various input services. You start out creating a `DemoInput` in the `default` section ❷. You'll add additional `case` sections in the following pages of this chapter.

Once the input service is constructed, you call its `Initialize` method ❸ to give the component a chance to load any game assets it requires. Finally, you call the input service's `Start` method ❹, allowing it to perform any tasks it requires to begin listening for user input.

At this point, your game's main page is complete, allowing the user to choose which input service to play the game with, and sending that choice to the game page. You're still lacking any real input services, and your game still only runs the demonstration script. The first of the input services you're going to implement is a virtual thumbstick that mimics the behavior of a real Xbox game pad.

15.2.2 Creating a thumbstick

Thumbsticks are a common feature of most game controllers. You can't use a game controller with the Windows Phone, but you can create a virtual thumbstick. The thumbstick will be drawn to the screen and the player will control the game by touching the screen and dragging the thumbstick in the desired direction. To capture the touches you're going to use the Touch API provided by the `TouchPanel` class. The `TouchPanel` provides methods and properties used to determine touch and gesture information. We'll focus on touch locations in this section and will cover how `TouchPanel` can be used with gestures later in the chapter.

The `GetState` method is used to ask the `TouchPanel` if the player is currently touching the screen. `GetState` returns a collection of `TouchLocations`, one for each point where the screen is currently being touched. If the player is touching the screen with only one finger, then only one `TouchLocation` is returned. If the player is touching the screen with three fingers, then three `TouchLocations` are returned.

> **NOTE** The maximum number of touches supported by Windows Phone is four. The maximum number of supported touches can be confirmed using `TouchPanel`'s `GetCapabilities` method. `GetCapabilities` returns a `TouchPanelCapabilities` structure that exposes a `MaximumTouchCount` property.

Let's consider the lifecycle of a touch. A player initiates a touch by pressing a finger to the screen. The player may hold the touch by continuing to press in the same location. The player may drag their finger across the screen to a new location. Finally, the player releases the touch by removing their finger from the screen. The `TouchLocation` structure reports the player's actions with the `State` property.

The `State` property is a `TouchLocationState` enumeration and will have the value of

- `Pressed` when the player initiates the touch
- `Moved` when the player holds in the same location or drags to a new location
- `Released` when the player removes their finger from the screen

For any given touch, only one press and one release will be reported. Any number of moves may be reported. Presses, releases, and moves are linked together with `TouchLocation`'s `Id` property. The location, in screen coordinates, of the touch is reported in `TouchLocation`'s `Position` property.

You can see `TouchLocations` in action by implementing a virtual thumbstick component. The thumbstick will implement `IGamePlayInput`. Add the component by creating a new project item and naming it `ThumbstickInput`. Implement the `IGamePlayInput` properties with automatic properties with private setters:

```
class TumbstickInput : IGamePlayInput
{
    public bool MoveForward { get; private set; }
    public bool MoveBackward { get; private set; }
    public bool TurnLeft { get; private set; }
    public bool TurnRight { get; private set; }
}
```

Before jumping into the thumbstick implementation, let's review the behavior and characteristics of a thumbstick, which are diagramed in figure 15.2. A thumbstick starts out in a rest position, and when pushed, moves around a central point. Because XNA drawing methods require a top left coordinate, you define the rest position as an offset of the center point.

To initiate movement, the player must touch the thumbstick and push it in one direction or another. If the initial touch isn't close enough to the thumbstick, it should be ignored. With your virtual thumbstick, you'll use a bounding box to filter out initial touches that are too far from the thumbstick. The *sensitivity area*, or *dead zone*, is used to account for the thickness of a fingertip. It's unlikely that the player will touch exactly in the center of the thumb sprite and you need a margin of error before moving the thumbstick.

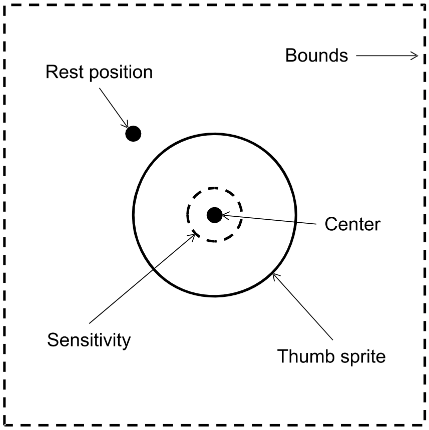


Figure 15.2 Thumbstick bounds, center point, rest position, and sensitivity area

Your implementation will need to account for each of the thumbstick characteristics. You need to define fields for the thumb sprite, center point and rest positions, and the sensitivity area:

```
Texture2D thumbsprite;
Vector2 centerPoint = new Vector2(115, 685);
Vector2 sensitivity = new Vector2();
Vector2 restPosition = new Vector2();
Vector4 bounds = new Vector4();
```

The thumbstick is drawn using a two-dimensional texture contained in the image file thumbstick.png. The image file is available with the source code from the book's website. Add the image file to the GraphicsWorldLibContent project, using the default content name *thumbstick*. Implement the component's `Initialize` method to load the texture and initialize the sensitivity fields. The sensitivity is initialized to be one quarter the size of the thumbstick. We've already stated that the bounding box is twice the height and width of the thumbstick. Now that you know the size of the sprite, you can calculate the coordinates of the bounding box. The bounding box coordinates also depend on the location of the center point. The following listing shows the implementation of `Initialize`, and how the sensitivity, resting position, and the bounding box coordinates are determined.

Listing 15.5 Initializing the thumbstick

```
public void Initialize(ContentManager content)
{
    thumbsprite = content.Load<Texture2D>("thumbstick");        ◁ ❶ Loading sprite texture

    sensitivity.X = thumbsprite.Width / 4;
    sensitivity.Y = thumbsprite.Height / 4;

    restPosition.X = centerPoint.X - thumbsprite.Width / 2;
    restPosition.Y = centerPoint.Y - thumbsprite.Height / 2;

    bounds.X = centerPoint.X - thumbsprite.Width;
    bounds.Y = centerPoint.Y - thumbsprite.Height;              ❷ Change bounding box coordinates
    bounds.Z = centerPoint.X + thumbsprite.Width;
    bounds.W = centerPoint.Y + thumbsprite.Height;
}
```

First you load the thumbstick texture using the `ContentManager` ❶. Next you adjust the `restPosition` to be offset from the center point by one half the width and height of the thumb sprite. Finally, you adjust the coordinates of the bounding box ❷ using both the new center point and the dimensions of the thumb sprite.

You're going to use the bounding box in the `Update` method to determine whether the player is pushing on the virtual thumbstick. The player starts pushing on the thumbstick by touching inside the bounding box. To keep the thumbstick from being pushed in two directions at once, you need to respond to only one touch point at a time. To track the touch point, you'll store the `Id` value of the `TouchLocation` in a field named `touchId`:

```
int touchId;
```

The `touchId` field's value will be zero when you're not tracking a touch point. Once you're tracking a touch point, the motion property values are set based on the touch location, as shown in the next listing.

Listing 15.6 Detecting thumbstick movement

```
public void Update()
{
    MoveForward = false;
    MoveBackward = false;
    TurnLeft = false;
    TurnRight = false;

    TouchCollection touches = TouchPanel.GetState();
    for (int index = 0; index < touches.Count; index++)
    {
        TouchLocation location = touches[index];
        if (location.State == TouchLocationState.Pressed
            && touchId == 0
            && location.Position.X > bounds.X
            && location.Position.X < bounds.Z
            && location.Position.Y > bounds.Y
            && location.Position.Y < bounds.W)
        {
            touchId = location.Id;
        }
        else if (location.Id == touchId)
        {
            if (location.State == TouchLocationState.Moved)
            {
                MoveForward = location.Position.Y <
                    (centerPoint.Y - sensitivity.Y);
                MoveBackward = location.Position.Y >
                    (centerPoint.Y + sensitivity.Y);
                TurnLeft = location.Position.X <
                    (centerPoint.X - sensitivity.X);
                TurnRight = location.Position.X >
                    (centerPoint.X + sensitivity.X);
            }
            else if (location.State == TouchLocationState.Released)
            {
                touchId = 0;
            }
        }
    }
}
```

① **Touch initiated inside bounding box**

② **Move only when outside sensitivity area**

Reset touchId when released

After resetting the motion properties, the `Update` method loops over each of the touch points. If you're not currently tracking a touch point and the current touch point is within the bounding box ①, then start tracking the touch point and remember the `TouchLocation`'s `Id` value. If you're tracking a touch point and the `TouchLocation` is in the `Moved` state, update the motion properties. The motion properties are set to `true` when the touch location is outside the sensitivity area ②. You check the sensitivity area to prevent the thumbstick from suddenly jumping when the player's touch is slightly off center. You don't check the bounding box after you start tracking the touch, as you don't want to penalize the player if they end up outside the bounding box while pressing the thumbstick.

It's not sufficient to respond to the player's touch. You need to draw the thumbstick and provide feedback based on how the player is pushing the controller. Before you can implement the `Draw` method, you need a `SpriteBatch`. You'll pass a `SpriteBatch` from the `GamePage` via the class constructor:

```
SpriteBatch spriteBatch;

public TumbstickInput(SpriteBatch batch)
{
    spriteBatch = batch;
}
```

The `Draw` method, implemented in the following listing, uses the `restPosition` as the initial point for drawing the thumbstick image. Within the `Draw` method, the motion properties are used to adjust the thumbstick drawing coordinates.

Listing 15.7 Drawing the thumbstick

```
public void Draw()
{
    Vector2 position = restPosition;
    if (MoveForward)
        position.Y -= thumbsprite.Height / 6;
    if (MoveBackward)
        position.Y += thumbsprite.Height / 6;
    if (TurnLeft)
        position.X -= thumbsprite.Width / 6;
    if (TurnRight)
        position.X += thumbsprite.Width / 6;

    spriteBatch.Begin();
    spriteBatch.Draw(thumbsprite, restPosition, Color.Black);
    spriteBatch.Draw(thumbsprite, position, Color.White);
    spriteBatch.End();
}
```

Adjust thumbstick position

Draw thumbstick base

The thumbstick position is moved by a value equal to one-sixth of the sprite's height and width. This value was chosen by trial and error until a value was found that gave the best player experience. An all-black copy of the sprite is drawn first, providing a visual cue to the player identifying the thumbstick's base.

The `ThumbstickInput` component is now complete. Now add the component to the game. You add code to the `switch` statement in the `OnNavigatedTo` method to construct and initialize a new instance of `ThumbstickInput`:

```
case "1":
    input = new ThumbstickInput(spriteBatch);
    break;
```

Your game is now ready to run. Launch the game from Visual Studio, choose the thumbstick option, and press Play. The game can be played with a virtual thumbstick that the player can use to move around in the 3D world. The thumbstick is the first of four input services you're going to implement in this chapter. The game has been built so that adding additional input mechanisms will be easy, and the next input service you implement is a button pad, another control surface found on an Xbox game pad.

15.2.3 Creating a button pad

In addition to thumbsticks, Xbox game controllers provide a set of buttons a player can use in games. In this section, you're going to build an input service that provides virtual A, B, X, and Y buttons the player can use to control the game. The button input service will use the Mouse API to respond to player input.

The XNA Framework originally provided the Mouse API to allow Windows-based games to be controlled by a mouse. Obviously, Windows Phone doesn't have a mouse, but the Mouse API can still be used by Windows Phone games. The framework translates touch input into a `MouseState` structure, which is retrieved with the `Mouse.GetState` method. The `MouseState` structure has several properties, most of which don't return any valuable information for the Windows Phone. The three properties that are useful are

- `LeftButton`
- `X`
- `Y`

`LeftButton` returns either `ButtonState.Pressed` or `ButtonState.Released`. The `X` and `Y` integer coordinates will only be valid when `LeftButton`'s value is `Pressed`. The Mouse API only works with the first touch point and can't be used to determine whether the player is touching the screen in multiple locations

You'll use the Mouse API in your button pad input service. The input service will check the `MouseState` in the update loop. If the left button is pressed, the coordinates are examined to see whether the player is touching one of the virtual buttons.

Create a new game component named `ButtonInput` and extend `IGamePlayInput`. Implement the `IGamePlayInput`'s motion properties with automatic properties. You also need to add a `SpriteBatch` field as you did for the thumbstick input component.

The images that form the buttons all come from the same file, or sprite sheet. The sprite sheet is shown in figure 15.3.

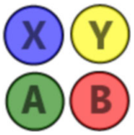


Figure 15.3 The button sprite sheet

To help with the drawing code, you'll create constants to hold sprite sheet positions:

```
static readonly Rectangle aSrc = new Rectangle(0, 75, 75, 75);
static readonly Rectangle bSrc = new Rectangle(75, 75, 75, 75);
static readonly Rectangle xSrc = new Rectangle(0, 0, 75, 75);
static readonly Rectangle ySrc = new Rectangle(75, 0, 75, 75);
```

The `ButtonInput` implementation will draw each of the buttons relative to a center point:

```
static readonly Point centerPoint = new Point(320, 640);
```

You also need several fields that will be used for drawing the buttons:

```
Rectangle aPosition = new Rectangle(0, 0, 75, 75);
Rectangle bPosition = new Rectangle(0, 0, 75, 75);
Rectangle xPosition = new Rectangle(0, 0, 75, 75);
Rectangle yPosition = new Rectangle(0, 0, 75, 75);

Rectangle aPosition2 = new Rectangle(0, 0, 65, 65);
Rectangle bPosition2 = new Rectangle(0, 0, 65, 65);
Rectangle xPosition2 = new Rectangle(0, 0, 65, 65);
Rectangle yPosition2 = new Rectangle(0, 0, 65, 65);
```

The button positions are calculated in the `Initialize` method, detailed in the next listing.

Listing 15.8 Initializing the drawing positions for each button

```
public void Initialize(ContentManager content)
{
    spriteSheet = content.Load<Texture2D>("buttons");

    aPosition.X = centerPoint.X;
    aPosition.Y = centerPoint.Y + 80;
    aPosition2.X = aPosition.X + 5;
    aPosition2.Y = aPosition.Y + 5;

    bPosition.X = centerPoint.X + 80;
    bPosition.Y = centerPoint.Y;
    bPosition2.X = bPosition.X + 5;
    bPosition2.Y = bPosition.Y + 5;

    xPosition.X = centerPoint.X - 80;
    xPosition.Y = centerPoint.Y;
    xPosition2.X = xPosition.X + 5;
    xPosition2.Y = xPosition.Y + 5;

    yPosition.X = centerPoint.X;
    yPosition.Y = centerPoint.Y - 80;
    yPosition2.X = yPosition.X + 5;
    yPosition2.Y = yPosition.Y + 5;
}
```

① **Load the sprite sheet**

You start by loading the button pad sprite sheet using the `ContentManager` ①. The sprite sheet will be stored in a `Texture2D` field named `spriteSheet`. Next each of the position rectangles need their x and y coordinates updated, and the new values are calculated as offsets from the center point. The A and Y buttons are moved above and below the center point, respectively. The X and B buttons are moved to the left and right of the center point, respectively. The `position2` rectangles will be used in the `Draw` method to provide a visual cue to the player that a button is pressed.

Now you're ready to update the motion properties. The `Update` method, shown in the following listing, reads the `MouseState` and uses the mouse location to determine the motion property values.

Listing 15.9 Determining input state from `MouseState`

```
public void Update()
{
    MouseState location = Mouse.GetState();

    if (location.LeftButton == ButtonState.Pressed)
    {
        MoveForward = yPosition.Contains(location.X, location.Y);
        MoveBackward = aPosition.Contains(location.X, location.Y);
        TurnLeft = xPosition.Contains(location.X, location.Y);
        TurnRight = bPosition.Contains(location.X, location.Y);
    }
    else
    {
```

If user is touching screen

```
        MoveForward = false;
        MoveBackward = false;
        TurnLeft = false;
        TurnRight = false;
    }
}
```

When the mouse is pressed, and the location is inside one of the button position rectangles, the corresponding motion property is set to `true`. With the motion properties set, you're now ready to draw the buttons. The `Draw` method, shown in listing 15.10, uses the sprite sheet, the position rectangles, and the motion properties to render buttons on the screen.

Listing 15.10 Drawing the button pad

```
public void Draw()
{
    spriteBatch.Begin();

    spriteBatch.Draw(spriteSheet, yPosition, ySrc, Color.White);
    if (MoveForward)
        spriteBatch.Draw(spriteSheet, yPosition2, ySrc, Color.White);

    spriteBatch.Draw(spriteSheet, aPosition, aSrc, Color.White);
    if (MoveBackward)
        spriteBatch.Draw(spriteSheet, aPosition2, aSrc, Color.White);

    spriteBatch.Draw(spriteSheet, xPosition, xSrc, Color.White);
    if (TurnLeft)
        spriteBatch.Draw(spriteSheet, xPosition2, xSrc, Color.White);

    spriteBatch.Draw(spriteSheet, bPosition, bSrc, Color.White);
    if (TurnRight)
        spriteBatch.Draw(spriteSheet, bPosition2, bSrc, Color.White);

    spriteBatch.End();
}
```

In addition to drawing each button, the `Draw` method draws a second, scaled-down copy of the button when its matching motion property is `true`. This button-within-a-button technique provides visual feedback that the button is pressed.

You're now ready to add the `ButtonInput` to the game. Just as you did with the `ThumbstickInput`, you construct the component in the `OnNavigatedTo` of the `GamePage`. You add code to the `switch` statement in the `OnNavigatedTo` method to construct and initialize a new instance of `ButtonInput`:

```
case "2":
    input = new ButtonInput(spriteBatch);
    break;
```

This wraps up the implementation of your `ButtonInput` that uses the Mouse API. Your next input service will use touch gestures.

15.2.4 Gaming with gestures

In this section, you're going to build an input service that allows the player to use touch-and-drag gestures to move about in the game. Touch-and-drag gestures are two of the gestures supported by the Gesture API. The Gesture API is provided by the XNA Framework to enable consistent multi-point touch gesture recognition for games.

The XNA Framework translates touch input into a `GestureSample` structure, which is retrieved with the `TouchPanel.ReadGesture` method. `GestureTypes` are defined as an enumeration, with the possible enumeration values as listed in table 15.1.

Table 15.1 Gesture types

Gesture type	Description
`DoubleTap`	Two quick touch and release motions in quick succession.
`DragComplete`	A drag motion has ended.
`Flick`	A touch followed by a quick swiping motion.
`FreeDrag`	A touch followed by movement in any direction.
`Hold`	A touch that's not immediately released.
`HorizontalDrag`	A touch followed by horizontal movement.
`Pinch*`	Two touches followed by movement bringing the two points closer together or farther apart.
`PinchComplete`	A pinch motion has ended.
`Tap`	A quick touch and release.
`VerticalDrag`	A touch followed by vertical movement.

*Pinch is the only gesture that uses multiple touch points.

Gesture processing is a compute-intensive operation and is turned off by the default. Gesture processing is only enabled for the gestures enabled with `TouchPanel`'s `EnabledGestures` property. `EnabledGestures` can be set to one or more of the `GestureType` values. `ReadGesture` will only return gestures that have been enabled.

The number and types of gestures are dependent on the types of gestures that have been enabled. For example, if only `Tap` has been enabled, two rapid touches will produce two distinct `Tap` `GestureSamples`. But if `Tap` and `DoubleTap` have both been enabled, two rapid touches will produce one `Tap` `GestureSample`, and one `DoubleTap` `GestureSample`. If `FreeDrag` is enabled by itself, a horizontal drag by the player will produce a `FreeDrag` `GestureSample`. But if both `FreeDrag` and `HorizontalDrag` are enabled, the same horizontal drag will only produce a `HorizontalDrag` `GestureSample`—no `FreeDrag` `GestureSample` will be read.

GestureSample defines the following properties:

- `Delta` and `Delta2`
- `GestureType`
- `Position` and `Position2`
- `TimeStamp`

The position properties provide coordinates of the touch points underlying the gesture. The delta properties provide clues to how far a touch has moved since the last time the same gesture was read. `Position2` and `Delta2` are only used in `Pinch` gestures, as `Pinch` is the only gesture to use multiple touch points. `Position` and `Delta` aren't reported for the `DragComplete` and `PinchComplete` `GestureSamples`.

Your gesture input service is going to use `HorizontalDrag` gestures to trigger left and right turns, and `VerticalDrag` gestures for forward and backward motion. Create a new game component named `GestureInput` and have it extend `IGamePlayInput`. Implement the `IGamePlayInput`'s motion properties with automatic properties. You also need to initialize adding a `SpriteBatch` field in the class constructor, as you did for the other input components.

The best time to turn on and off gesture processing is when the gesture input component is started and stopped. You'll implement `IGamePlayInput`'s `Start` and `Stop` methods, as shown in the following listing, to enable and disable gesture processing.

Listing 15.11 Enabling and disabling gestures

```
using Microsoft.Xna.Framework.Input.Touch;

public void Start()
{
    TouchPanel.EnabledGestures = GestureType.HorizontalDrag
        | GestureType.VerticalDrag | GestureType.DragComplete;
}

public void Stop()
{
    TouchPanel.EnabledGestures = GestureType.None;
}
```

Turn on drag gestures ①

② **Turn off gesture processing**

When the `GestureInput` is started, you set the `TouchPanel`'s `EnabledGestures` property to a binary OR of `HorizontalDrag`, `VerticalDrag`, and `DragComplete` ①. When the input service is stopped, you turn off gesture processing ② by setting `EnabledGestures` to `None`.

With the desired gestures enabled, you can now call `ReadGesture`. In the game, gestures are read and handled in the `Update` method, as implemented in the next listing.

Listing 15.12 Detecting gestures

```
Vector2 touchPosition;

public override void Update(GameTime gameTime)
{
    while (TouchPanel.IsGestureAvailable)
    {
```

① **Check for available gestures**

```
        GestureSample gs = TouchPanel.ReadGesture();
        touchPosition = gs.Position;
        switch (gs.GestureType)
        {
            case GestureType.HorizontalDrag:
                TurnLeft = gs.Delta.X < 0;
                TurnRight = gs.Delta.X > 0;
                break;

            case GestureType.VerticalDrag:
                MoveForward = gs.Delta.Y < 0;
                MoveBackward = gs.Delta.Y > 0;
                break;

            default:
                TurnLeft = false;
                TurnRight = false;
                MoveForward = false;
                MoveBackward = false;
                break;
        }
    }
    base.Update(gameTime);
}
```

Remember touch coordinates

② Stop moving on DragComplete

The `IsGestureAvailable` property should be used to check whether a gesture is ready to be read. If you call `ReadGesture` when `IsGestureAvailable` is `false`, an `InvalidOperationException` is thrown. Multiple gestures could be available and you continue to loop until all gestures are read ❶. When drag gestures are read, you look at the `Delta` property to determine whether the motion properties should be set to `true`. When a `DragComplete` gesture is read ❷, you reset all the motion properties to `false`.

Like your other input components, `GestureInput` draws a visual cue on the screen when the player is moving. The visual cue is an arrow that follows the player's touch-and-drag gestures. The arrows are read from a file named directional.png, which you add to the GraphicsWorldLibContent project. Implement the `GestureInput`'s `Initialize` method to load the sprite sheet into a field named `spriteSheet`:

```
public void Initialize(ContentManager content)
{
    spriteSheet = content.Load<Texture2D>("directionals");
}
```

With the sprite sheet loaded and the touch position stored, drawing is easy. The following listing shows the `Draw` method code used to draw the arrows that appear to follow the player's touch.

Listing 15.13 Drawing

```
static readonly Rectangle forwardSrc = new Rectangle(0, 0, 50, 50);
static readonly Rectangle backwardSrc = new Rectangle(50, 50, 50, 50);
static readonly Rectangle leftSrc = new Rectangle(0, 50, 50, 50);
static readonly Rectangle rightSrc = new Rectangle(50, 0, 50, 50);
```

```
public void Draw()
{
    spriteBatch.Begin();

    if (MoveForward)
        spriteBatch.Draw(spriteSheet, touchPosition,
            forwardSrc, Color.Red);

    if (MoveBackward)
        spriteBatch.Draw(spriteSheet, touchPosition,
            backwardSrc, Color.Red);

    if (TurnLeft)
        spriteBatch.Draw(spriteSheet, touchPosition, leftSrc, Color.Red);

    if (TurnRight)
        spriteBatch.Draw(spriteSheet, touchPosition, rightSrc, Color.Red);

    spriteBatch.End();
}
```

You're now ready to add the `GestureInput` to the game. Just as you did with the other input services, you construct the component in the `OnNavigatedTo` of `GamePage`. You add code to the `switch` statement in the `OnNavigatedTo` method to construct and initialize a new instance of `GestureInput`:

```
case "3":
    input = new GestureInput(spriteBatch);
    break;
```

You now have three different input services in your game. All three are touch-based services, each using a different API to obtain touch and position information. The last input service we cover in this chapter isn't a touch-based implementation and uses the motion sensor instead.

15.2.5 Moving with the motion sensor

You've implemented three of the four input services required for the game. The last input service makes use of the `Motion` class, moving and turning the player as the phone is tilted. When the phone is tilted side-to-side, the in-game player will be turned to the left or right. When the phone is tilted front-to-back, the player will move forward or backward.

The Motion API was covered in depth in chapter 8. For this game, you'll keep the usage of the motion sensor simple, and will use the poll-based `CurrentValue` property instead of the `CurrentValueChanged` events.

Some phones don't support a motion sensor, so you need to prevent the user from selecting the motion sensor option in the main page. You disable the motion sensor `RadioButton` shown on the main page from the `MainPage` constructor. You set the `IsEnabled` property to the value returned by the motion sensor's `IsSupported` static method:

```
motionSensor.IsEnabled = Motion.IsSupported;
```

Now you're ready to implement the new input service. Create a new game component named `MotionInput` and have it extend `IGamePlayInput`. Implement the `IGamePlayInput`'s motion properties with automatic properties. You also need to add a `SpriteBatch` field as you did for the other game components, initializing the field in the class constructor. Add a field to store an instance of the motion sensor:

```
Motion motionSensor = new Motion();
```

The motion sensor must be started and stopped, which you'll perform in the input service's `Start` and `Stop` methods. When a game's only interaction with the player is via the motion sensor, special care must be taken to prevent the phone from locking. The lock screen is triggered when the user hasn't touched the screen in a specified amount of time. When a user is playing a motion sensor-based game, they're not touching the screen, and their game will pause as the phone enables the lock screen.

You can prevent a locked screen by disabling `UserIdleDetectionMode` through the `PhoneApplicationService`. `UserIdleDetectionMode` should be disabled when the motion sensor component is started, and enabled again when the input service is stopped. The next listing shows the `Start` and `Stop` method implementations that start and stop the motion sensor, as well as enable and disable `UserIdleDetectionMode`.

Listing 15.14 Starting and stopping motion sensor input

```
public void Start()
{
    motionSensor.Start();
    PhoneApplicationService.Current.UserIdleDetectionMode
        = IdleDetectionMode.Disabled;
}

public void Stop()
{
    motionSensor.Stop();
    PhoneApplicationService.Current.UserIdleDetectionMode
        = IdleDetectionMode.Enabled;
}
```

The input component will record a turn when the left or right edge of the screen is tilted down. Tilting the top or bottom edge will move the player forward and backward. To determine whether the device is tilted, you use the `Gravity` vector from the `MotionReading` class. The following listing shows how you'll detect motion in the `Update` method and adjust the player's position.

Listing 15.15 Updating player movement

```
readonly float minimumVector = (float)Math.Sin(MathHelper.PiOver4 / 3);
readonly float maximumVector = (float)Math.Sin(MathHelper.PiOver4);

public void Update()                                        Tilt thresholds ❶
```

```
{
    var y = motionSensor.CurrentValue.Gravity.Y;
    MoveBackward = y < -minimumVector && y >= -maximumVector;
    MoveForward = y > minimumVector && y < maximumVector;

    var x = motionSensor.CurrentValue.Gravity.X;
    TurnRight = x > minimumVector && x <= maximumVector;
    TurnLeft = x < -minimumVector && x >= -maximumVector;
}
```

② **Check top/ bottom edge tilt**

③ **Check side edge tilt**

To prevent the device from being too sensitive to movement, you only update the player's position when the device is tilted between 15 and 45 degrees. You define two constants that represent the length of `X` and `Y` `Gravity` vectors corresponding to the tilt thresholds ① in either direction. Next you obtain the `Y` `Gravity` vector to determine how much the top and bottom edge is tilted ②. If the `Y` `Gravity` vector is between the minimum and maximum thresholds, you set `MoveBackward` or `MoveForward` to true. Finally, you set `TurnLeft` and `TurnRight` using the `X` `Gravity` vector ③.

When the player has tilted the phone enough to trigger motion, you'll draw an arrow on the screen in the direction of the motion. You're going to use the same sprite sheet to draw arrows that you used in the `GestureInput`. To help with the drawing code you define some fields:

```
static readonly Rectangle forwardSrc = new Rectangle(0, 0, 50, 50);
static readonly Rectangle backwardSrc = new Rectangle(50, 50, 50, 50);
static readonly Rectangle leftSrc = new Rectangle(0, 50, 50, 50);
static readonly Rectangle rightSrc = new Rectangle(50, 0, 50, 50);

Vector2 forwardPos = new Vector2(215.0f, 82.0f);
Vector2 backwardPos = new Vector2(215.0f, 745.0f);
Vector2 leftPos = new Vector2(10.0f, 375.0f);
Vector2 rightPos = new Vector2(425.0f, 375.0f);
```

The arrows will be drawn at the center point of the edge that corresponds to the triggered motion. For example, an arrow will be drawn in the center of the top edge, just below the scoreboard, when moving forward. The sprite sheet containing the arrow images is loaded in the `Initialize` method:

```
public void Initialize(ContentManager content)
{
    spriteSheet = content.Load<Texture2D>("directionals");
}
```

With the sprite sheet loaded and the target positions stored, drawing is straightforward. The next listing shows the `Draw` method code to draw the arrows that appear when motion is triggered.

Listing 15.16 Drawing

```
public void Draw()
{
    spriteBatch.Begin();

    if (MoveBackward)
        spriteBatch.Draw(spriteSheet, forwardPos, forwardSrc, Color.Red);
    if (MoveBackward)
        spriteBatch.Draw(spriteSheet, backwardPos, backwardSrc, Color.Red);

    if (TurnLeft)
        spriteBatch.Draw(spriteSheet, leftPos, leftSrc, Color.Red);

    if (TurnRight)
        spriteBatch.Draw(spriteSheet, rightPos, rightSrc, Color.Red);

    spriteBatch.End();
}
```

You're now ready to add the `MotionInput` to the game. Just as you did with the other input services, you need to add a case section to the `switch` statement in the `GamePage.OnNavigatedTo` method:

```
case "4":
    input = new MotionInput(spriteBatch);
    break;
```

With the completion of `MotionInput` you're finished with the game. The player can now choose between using a thumbstick, a button pad, dragging a finger across the screen, or tilting the phone.

In this section we looked at the different APIs that can be used to receive game input from a player. The mouse is suitable if you only support one touch point and only need to know when and where the player is touching the screen. Gestures are useful when your game is designed to work with the most common set of touch operations—`Tap`, `Flick`, `Drag`, and `Pinch`. You must resort to raw touch handling if you use more complex touch-processing logic.

What's important isn't that you can make virtual thumbsticks or button pads. The important thing is to understand that there are different ways you can use touch and the motion sensor to build a game. Knowing how the different input systems work will help you decide which method best fits your own game.

15.3 Summary

The XNA Framework provides a few different APIs for handling touch input. You used the `TouchPanel` class to access raw touch input. The `TouchPanel` also converts raw touches into predefined gestures such as Tap, Double Tap, Drag, and Pinch. The Mouse API can also be used in place of raw touch. You used each of these APIs to build components that translated touch input into player motion. You added options to the game enabling the user to change which of the input components to use during game play.

Though input handling was the main focus of the chapter, we covered several other topics. We showed you how to draw images and textures with the `SpriteBatch`. You combined several images into a sprite sheet. You also learned how to use a Silverlight button within an XNA game loop to resume a paused game.

Supporting pause in your game is important, because the operating system might interrupt a game to handle other tasks such as a phone call. When the game is reactivated from an interruption, the game should restart in a paused state. The player should be able to pause a running game by pressing the Back button. We showed how to detect a Back button press and determine whether the game should be paused or whether the game should exit.

This chapter wraps up our coverage of Windows Phone 7. Early in the book we introduced you to programming concepts unique to Windows Phone. We looked at the application lifecycle, how to participate in fast application switching, and how to recover from tombstoning. We built background agents and created alarms and reminders. We then looked at how to use launchers and choosers to access the phone dialer, email, and text messaging. Next you learned how to interact with the phone's hardware and sensors including the camera, accelerometer, compass, gyroscope, and motion sensor.

Windows Phone 7 is an exciting platform for application and game development. Silverlight and XNA can be used together to build exciting games and applications. The built-in applications and hubs provide extension points allowing creative developers the opportunity to extend and enhance the Windows Phone experience. We showed you how to extend the Pictures Hub and the Music + Videos Hub. Your application can also pin Live Tiles on the start screen, giving users glance and go information and quick access to your application.

You now have the tools necessary to create your own Windows Phone games and applications that you can sell in the Application Marketplace. If you haven't signed up with the AppHub, you'll need to do so before you can publish to the Marketplace. You can read more about the Marketplace in appendix C.

appendix A
Microsoft Expression Blend for Windows Phone

Throughout this book you've used Visual Studio Express for Windows Phone to build sample applications. Visual Studio is a great tool for writing and debugging C# and Visual Basic code, but is only moderately useful for working with XAML documents. The Visual Studio XAML designer allows you to drag and drop controls onto a page and provides property editors to allow you to manipulate the appearance of the user interface. The Visual Studio designer is built for software developers.

Visual Studio isn't so friendly for the members of your team who focus on visual design instead of code. Microsoft Expression Blend allows visual designers to create user interfaces without writing a single line of C# or Visual Basic code. Designers can work on user interfaces using the same project files that the developer uses in Visual Studio. Expression Blend was installed as part of the Windows Phone SDK 7.1.

In this appendix you'll learn how to create a new project in Microsoft Expression Blend. You'll use the tools and designers provided by Expression Blend to add an ellipse to the application and change the ellipse's look and feel. Finally you'll use Silverlight Animation and make the ellipse act like a bouncing ball using animation and behaviors when the user touches the screen. Before you create the sample application, you need to know a bit more about Expression Blend's tools and designers.

A.1 Expression Blend's tools and designers

Like Visual Studio, the Expression Blend user interface is made up of a workspace divided into a number of dockable panels surrounding the visual designer and text editors. Figure A.1 shows the Expression Blend user interface with a sample project loaded. The dockable panels include various tools such as the Objects and Timeline tool, the Properties tool, and the Projects tool.

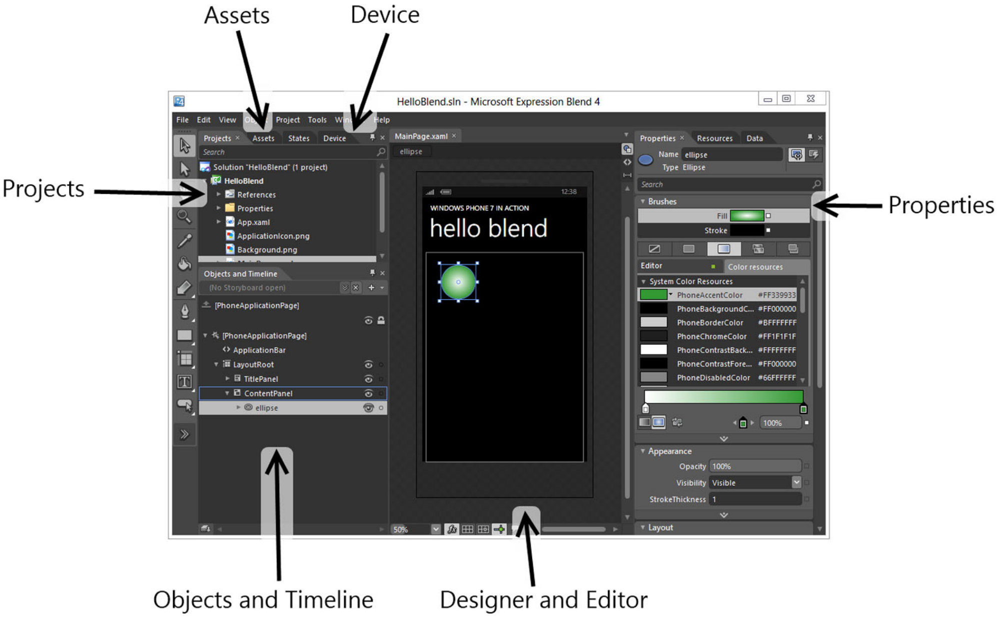

Assets
Device
Projects
Properties
Objects and Timeline
Designer and Editor
HelloBlend.sln - Microsoft Expression Blend 4
File Edit View Object Project Tools Window Help
Projects Assets States Device
MainPage.xaml
Properties Resources Data
Solution 'HelloBlend' (1 project)
HelloBlend
References
Properties
App.xaml
ApplicationIcon.png
Background.png
Objects and Timeline
(No Storyboard open)
[PhoneApplicationPage]
ApplicationBar
LayoutRoot
TitlePanel
ContentPanel
ellipse
WINDOWS PHONE 7 IN ACTION
hello blend
Name ellipse
Type Ellipse
Brushes
Fill
Stroke
System Color Resources
PhoneAccentColor #FF339933
PhoneBackgroundC... #FF000000
PhoneBorderColor #BFFFFFFF
PhoneChromeColor #FF1F1F1F
PhoneContrastBack... #FFFFFFFF
PhoneContrastFore... #FF000000
PhoneDisabledColor #66FFFFFF
Appearance
Opacity 100%
Visibility Visible
StrokeThickness 1
Layout

Figure A.1 The Expression Blend user interface showing the Projects, Assets, Device, Properties, and Objects and Timeline panels. The designer resides in the center of the workspace.

Some of the tools found in Expression Blend, such as the Projects tool, might be familiar to users of Visual Studio, because the two products provide overlapping features. The following list describes the tools shown in figure A.1:

- The Projects tool is similar to the Visual Studio Solution Explorer and shows all the code and content files that are part of the application. References to external libraries and assemblies are also shown.
- The Assets tool is similar to Visual Studio's Toolbox and lists the various controls, shapes, layout containers, and behaviors that are available for use. Items listed in the Assets Tool can be added to a page with a double-click or by drag-and-drop.
- The Device tool allows you to change the page orientation and the system theme color used by the designer when displaying a page. The Device Tool also provides the target device selector which allows you to choose whether the application should be deployed to the emulator or a connected phone.
- The Properties tool is similar to the Properties view in Visual Studio. The Properties tool allows you assign values to various control properties such as Fill, Background, Height, or Width.
- XAML and code documents are displayed by the designer or editor in the center of the workspace. The XAML editor supports three views—a visual designer, a code editor, and a split design/code view.

- At first glance, the Objects and Timeline view looks similar to Visual Studio's Document Outline view. But this useful tool provides several unique features for managing animations, which we'll demonstrate in a few pages.

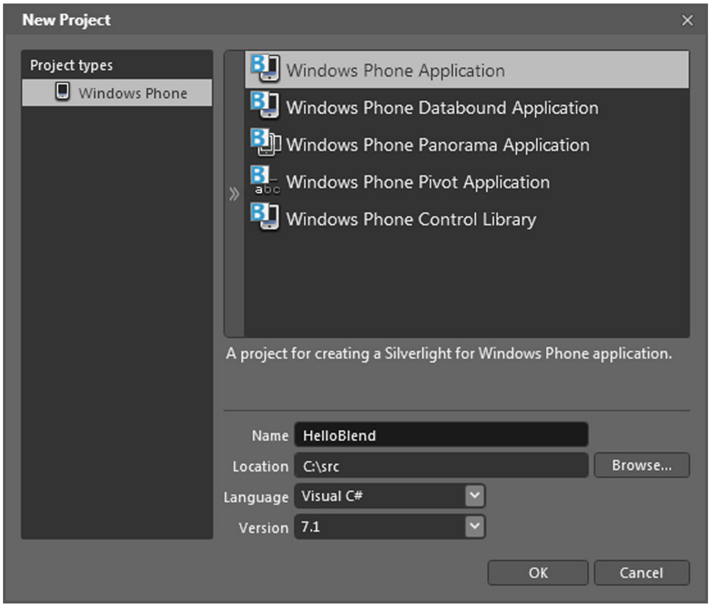
Now that you've been introduced to Expression Blend's tools, you'll create a sample application.

A.2 Creating an application

Creating a new application in Expression Blend is nearly identical to creating an application in Visual Studio. From either the splash screen or the file menu, selecting the New Project option will display the New Project dialog box. The New Project dialog box, shown in figure A.2, prompts you for a project template, application name, programming language, and Windows Phone SDK version.

Expression Blend provides five of the project templates provided by Visual Studio for creating Silverlight applications and control libraries. You can't use Expression Blend to create Silverlight with XNA or background agent projects.

To start your sample application, create a new project using the Windows Phone Application project template and name the application HelloBlend. Take a few minutes to explore Expression Blend and the tools you learned about in the previous section. Use the Objects and Timelines view to select the `PageTitle TextBlock` and the Properties panel to change the text from *page name* to *hello blend*. Next, switch to the Device panel, change the preview accent color to Green, and select the Windows Phone Emulator as the run target. Finally, run the project by pressing the F5 key or by selecting Run Project from the Project menu.


New Project
Project types
Windows Phone
Windows Phone Application
Windows Phone Databound Application
Windows Phone Panorama Application
Windows Phone Pivot Application
Windows Phone Control Library
Browse...
OK
Cancel

A project for creating a Silverlight for Windows Phone application.

Name HelloBlend
Location C:\src
Language Visual C#
Version 7.1

Figure A.2 Expression Blend's New Project dialog box listing the Windows Phone project templates

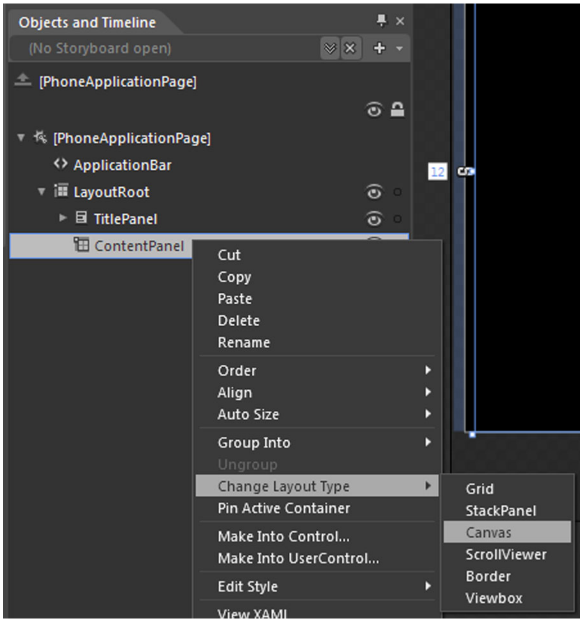

Objects and Timeline
(No Storyboard open)
[PhoneApplicationPage]
[PhoneApplicationPage]
ApplicationBar
LayoutRoot
TitlePanel
ContentPanel
12
Cut
Copy
Paste
Delete
Rename
Order
Align
Auto Size
Group Into
Ungroup
Change Layout Type
Pin Active Container
Make Into Control...
Make Into UserControl...
Edit Style
View XAML
Grid
StackPanel
Canvas
ScrollViewer
Border
Viewbox

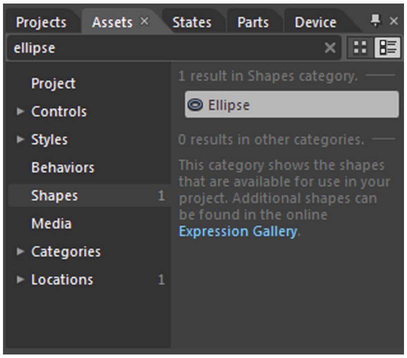
Figure A.3 Changing the `ContentPanel` from a `Grid` to a `Canvas`

A.3 Adding a shape to the page

Now that you have a running application built in Expression Blend, you'll use the Asset tool to add an ellipse to the page. Then you'll use the Properties view to assign a new gradient brush to the ellipse so that it looks more like a ball.

Before you add the ellipse to the page, you should change the `ContentPanel` from a `Grid` container to a `Canvas`. The Objects and Timeline tool makes it easy to implement this change with the Change Layout Type option on the context menu, shown in figure A.3.

Next, you're going to add an `Ellipse` to the `ContentPanel` using the Assets tool. Open the Assets tool and select the Shapes category. You can scroll through the list until you find `Ellipse`, or you can type *ellipse* into the search widget. The search widget quickly narrows down the list and displays only matching items in the Assets tool, as seen in figure A.4. With the `ContentPanel` selected in the Objects and Timeline view, double-click `Ellipse` in the Assets tool.


Projects
Assets
States
Parts
Device
ellipse
Project
Controls
Styles
Behaviors
Shapes 1
Media
Categories
Locations 1
Ellipse

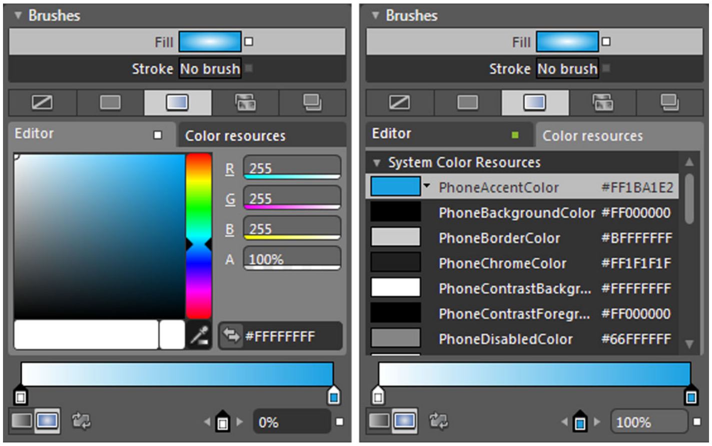
1 result in Shapes category.

0 results in other categories.

This category shows the shapes that are available for use in your project. Additional shapes can be found in the online Expression Gallery.

Figure A.4 The Assets view filtered to show only the `Ellipse`

By default, the new `Ellipse` shape will be a circle, which is what you want. You'll simulate a 3D lighting effect using a `GradientBrush`. Select the newly added element by clicking on Ellipse in the Objects and Timeline tool. An object's properties



Stroke No brush
R 255
G 255
B 255
A 100%
#FFFFFFFF
0%

Stroke No brush

PhoneAccentColor	#FF1BA1E2
PhoneBackgroundColor	#FF000000
PhoneBorderColor	#BFFFFFFF
PhoneChromeColor	#FF1F1F1F
PhoneContrastBackgr...	#FFFFFFFF
PhoneContrastForegr...	#FF000000
PhoneDisabledColor	#66FFFFFF

100%

Figure A.5 Specifying the `Ellipse`'s `Fill` using the brush editor. The left image shows the first gradient stop with white as the selected color. The right image shows the second gradient stop with the `PhoneAccentColor` resource as the selected color.

can be modified using the Properties tool, and in this case you want to change the `Fill` property.

Changing the `Fill` property can be confusing the first time you use Expression Blend's brush property editor. First you want to click on the fill line in the editor, seen in figure A.5. Next you want to specify a `GradientBrush` using the middle button in the toolbar. Then you'll specify a radial gradient using the button in the lower-left corner of the brush editor. The first gradient stop color will be white, whereas the second gradient stop color will use the `PhoneAccentColor` resource.

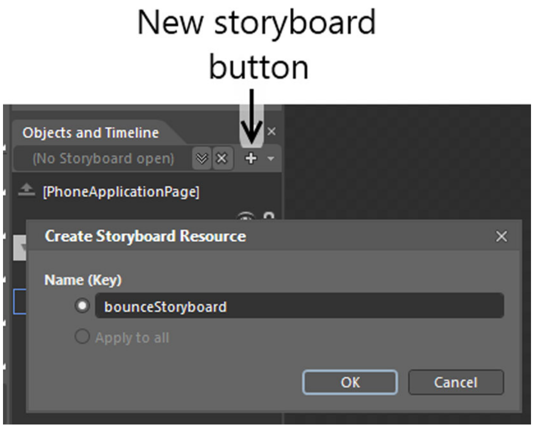
If you've configured the brush correctly, the ellipse should look more like a ball, with a whitish center and a solid color perimeter. You can check that you configured the brush to use the theme color by changing the preview accent color setting in the Device tool. Now try launching the application in the emulator—does the ellipse pick up the theme color specified in the settings application?

A.4 Animating the ellipse

You now have a simple application that displays what appears to be a ball on the screen. Your goal is to move the ball across the screen, bouncing off the edges as it moves. You'll implement the movement using Silverlight animation and storyboards. Expression Blend makes it easy to create storyboards and animate controls with the Object and Timeline tool.

You'll use the Objects and Timeline tool to create a new storyboard, and then add the bounce animations to the storyboard. Create a new storyboard by clicking the New Storyboard button, shown in figure A.6, and entering the name `bounceStoryboard` in the Create Storyboard Resource dialog box.

Once a storyboard is open, the Objects and Timeline view changes to show a timeline with playback controls. Now you're ready to record your bounce animation. Select the Ellipse in the Object tree, and click the Record Keyframe button, shown in figure A.7, to give your ball a starting point. Next move the playhead to the 1-second

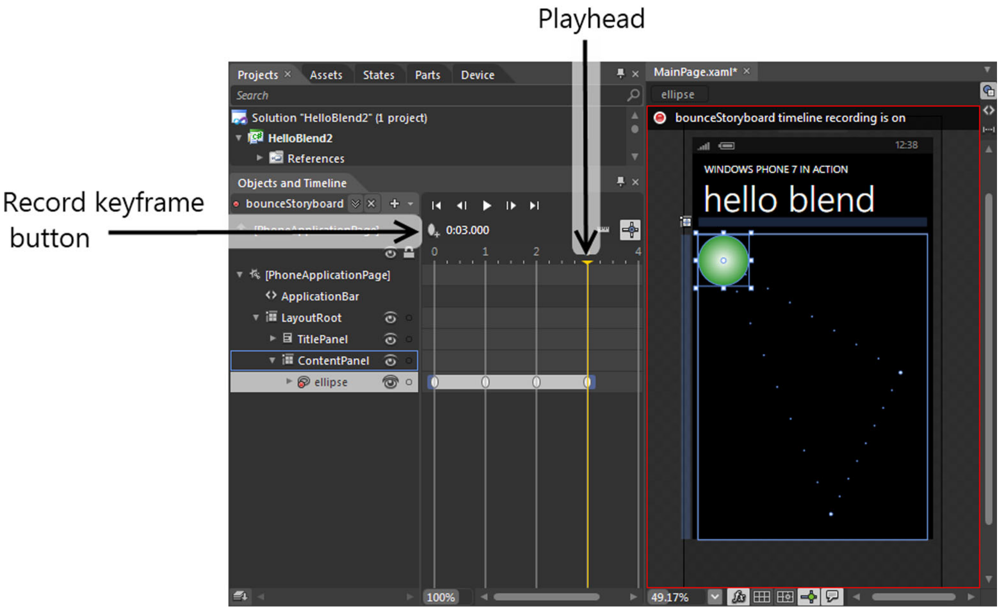

New storyboard button
Objects and Timeline
(No Storyboard open)
[PhoneApplicationPage]
Create Storyboard Resource
Apply to all
OK
Cancel

Name (Key): bounceStoryboard

Figure A.6 Create a new storyboard by clicking the New Storyboard button and entering a name in the Create Storyboard Resource dialog.

mark and drag the Ellipse to the bottom of the `ContentPanel`. Repeat the operation, setting the playhead and ellipse position for the 2-and 3-second marks. The completed storyboard is shown in figure A.7.

If you run the application now, nothing is different than before you created the bounce storyboard. The ball sits in the corner doing nothing. Next you'll add a trigger to play the animation when the user taps the screen.


Playhead
Record keyframe button
Projects
Assets
States
Parts
Device
Search
Solution "HelloBlend2" (1 project)
HelloBlend2
References
Objects and Timeline
bounceStoryboard
0:03.000
0
1
2
4
[PhoneApplicationPage]
ApplicationBar
LayoutRoot
TitlePanel
ContentPanel
ellipse
MainPage.xaml*
ellipse
12:38
WINDOWS PHONE 7 IN ACTION
hello blend
100%
49.17%

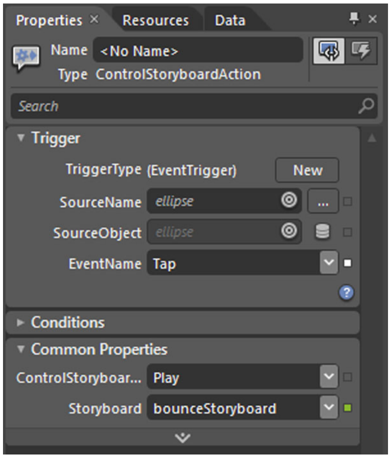
bounceStoryboard timeline recording is on

Figure A.7 The Objects and Timeline tool with the complete bounce storyboard

A.5 Triggering an animation

To start an animation, a developer using Visual Studio might subscribe to the `Ellipse`'s `Tap` event and call the storyboard `Begin` method from code behind. Expression Blend allows designers to perform the same task without writing code. Expression Blend makes this possible through a feature named *behaviors*. Behaviors are pre-built components that use Silverlight attached properties to bind to user interface controls and execute a predefined action.

You want the ball to bounce every time the user taps on it, so you'll use a behavior named `ControlStoryboardAction` to start the bounce storyboard you created in the last section. With the `Ellipse` selected in the Objects and Timeline tool, open the Assets tool to the Behaviors category and double-click `ControlStoryboardAction`. A new node should now exist in the Objects and Timeline tool. Go ahead and select the new node, and then open the Properties view. Figure A.8 shows the Properties view for the `ControlStoryboardAction`.



Name <No Name>
Type ControlStoryboardAction
TriggerType (EventTrigger)
SourceName ellipse
SourceObject ellipse
EventName Tap
ControlStoryboar... Play
Storyboard bounceStoryboard

Figure A.8 Setting the properties for the `ControlStoryboardAction`

You need to make changes to two of the `ControlStoryboardAction`'s properties. First, change the `EventName` property to `Tap`. Next change the `Storyboard` property to `bounceStoryboard`. Run the application now and tap on the ball on the screen. You should now see the ball move around on the screen while the animation plays.

A.6 Summary

Microsoft Expression Blend is a powerful application development tool and we've just scratched the surface in this appendix. Expression Blend provides support for custom controls, design time data and view model support for the Model-View-ViewModel (MVVM) pattern, and SketchFlow for rapid application prototyping. If you'd like to read more about Expression Blend, check out *Expression Blend in Action* by Joel Cochran, available from Manning Publications.

Microsoft provides tutorials, training videos, and starter kits on the Expression website (see http://mng.bz/M2Ll).

appendix B

Silverlight and the Extensible Application Markup Language

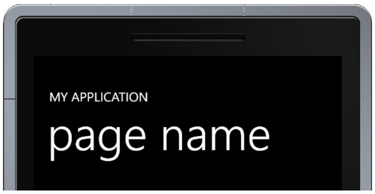
What's Silverlight? Silverlight is an implementation of Microsoft's .NET Framework that has been scaled down and tuned to build good-looking, interactive, and responsive client applications. Silverlight includes the .NET Framework APIs needed for client-side application features and strips out APIs not well suited for restricted environments like the browser and Windows Phone. Silverlight application user interfaces are designed with the *Extensible Application Markup Language (XAML)* and its related Silverlight class libraries.

XAML was first introduced as part of the Windows Presentation Foundation. XAML excels as a user interface markup language, separating the user interface design from the code behind implementing an application's business logic. XAML not only defines WPF and Silverlight UIs, but is one of the options when building WinRT applications for the upcoming Windows 8 operating system. If you're coming from a web development background, you can think of XAML as similar to HTML, which you use to create UI for your web page. XAML is an XML document that represents the hierarchical structure of an application's user interface.

This appendix serves as a quick introduction to XAML. We'll cover basic UI layout and the available UI controls. We'll move on to an introduction to the data binding features built into Silverlight. We wrap up using `DataTemplates` to create a user interface for a domain model object. Let's start by examining the default XAML generated when you create a new application with the Windows Phone

Application project template. Figure B.1 shows a screenshot of the application generated by the project template.

The default application generates a page that contains a `Grid` layout control, a `StackPanel` layout control, and two `TextBlock`s. The XAML markup for the form is shown in the following listing. We'll refer back to this listing several times in the next few pages.



MY APPLICATION

page name

Figure B.1 A screenshot of the default application

Listing B.1 XAML generated by the Windows Phone Application project template

```
<phone:PhoneApplicationPage x:Class="Primer.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone=
       "clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:primer="clr-namespace:Primer"
...
    />
    <Grid x:Name="LayoutRoot" Background="Transparent">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28"
           Orientation="Vertical" >
           <TextBlock x:Name="ApplicationTitle" Text="MY APPLICATION"
                Style="{StaticResource PhoneTextNormalStyle}"/>
           <TextBlock x:Name="PageTitle" Text="page name" Margin="9,-7,0,0"
                Style="{StaticResource PhoneTextTitle1Style}"/>
        </StackPanel>
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"/>
    </Grid>
</phone:PhoneApplicationPage>
```

Root element ❶

The root of the XAML document is an element of type `PhoneApplicationPage` ❶. Elements in a XAML document must be matched with a class name from either the Silverlight class libraries or your application's code. The root element also contains a `Class` attribute that declares the full name of the C# class containing the code behind for the user interface. The class named in the `Class` attribute must be derived from the class used as the root element. In listing B.1, the class `MainPage` is derived from the class `PhoneApplicationPage`.

You might have noticed that the root element uses the XML namespace `phone`. In XAML, namespaces can be declared with an assembly name or with a URI. An example of the URI version can be seen in the third line of listing B.1, declaring the `x` namespace. The fourth line of the listing shows the assembly name version declaring the `phone` namespace. The fifth line of the listing is the declaration for the `Primer` namespace, which is part of the application and doesn't need to specify an assembly name.

Most XAML documents you create for Windows Phone will derive from either `PhoneApplicationPage` or from `UserControl`. Both `PhoneApplicationPage` and `UserControl` expect to have a single child control. In most cases the child control will be a layout control derived from the `Panel` class.

B.1 Layout controls

Layout controls are containers of other controls and are responsible for automatically positioning their children on the screen. The Windows Phone SDK includes three different layout controls named `StackPanel`, `Grid`, and `Canvas`.

The `StackPanel` control lays out its children in a horizontal or vertical stack, depending on the value of its `Orientation` property. The `StackPanel` in listing B.1 specifies an orientation of `Horizontal`, which is the default value if an `Orientation` attribute isn't declared. A horizontal `StackPanel` stacks controls one on top of the other and will ensure that all child controls have exactly the same width. A vertical `StackPanel` stacks controls side by side and will ensure that all controls have exactly the same height.

The `Grid` control lays out its children in a series of rows and columns. By default, a `Grid` has only one row and one column. Additional rows and columns are specified with the `Grid.RowDefinitions` and `Grid.ColumnDefinitions` properties, respectively. The `Grid` control named `LayoutRoot` in listing B.1 has two rows and a single column. The first row is given a height of `Auto` and uses as much height as needed. The second row is given a height of `*`, which tells the `Grid` to give the row a height that fills all of the remaining space. Child controls specify their row and column with `Grid.Row` and `Grid.Column` attributes. For example, the `Grid` named `ContentPanel` specifies that it should be placed in the second row. Row and column indexes are zero-based. A control can span multiple rows and columns using the `Grid.RowSpan` and `Grid.ColumnSpan` attributes. If a child control doesn't specify its row and column values, it'll be placed in the first row and first column.

The `Canvas` control lays out its children using absolute positioning. The positions of child elements are declared using `Canvas.Left` and `Canvas.Top` properties. If a child doesn't declare its position, it'll be placed at coordinate (0, 0).

B.2 Interacting with Silverlight controls

Silverlight for Windows Phone contains many of the common controls you'd expect in a user interface library. You should be aware that some of the controls present in WPF and Silverlight for the browser aren't supported on the Windows Phone. Check the MSDN documentation for a full list of controls supported by Windows Phone.

Let's take a look at how to declare a simple form with a few controls and how to interact with the controls from code. The form will contain a `TextBlock` to display a label, a `TextBox` to receive input from the user, and a `Button`:

```
<StackPanel>
    <TextBlock Text="Please enter your name" />
    <TextBox x:Name="nameTextBox" />
    <Button Content="Save" Width="150" Click="Button_Click" />
</StackPanel>
```

You've given the `TextBox` a name using the `x:Name` attribute. The compiler will automatically generate a field for named controls, allowing you to easily access the control from code behind. The `Button` control is defined with a `Width` value of 150 pixels. When the button is tapped by the user, a `Click` event is raised, and a custom event handler method named `Button_Click` is called:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    string name = nameTextBox.Text;
    MessageBox.Show("You entered : " + name);
}
```

In the `Button_Click` method, you use the generated field `nameTextBox` to retrieve the text entered by the user. You then use the `MessageBox` class to display a message to the user with a pop-up window.

B.3 Styles and resources

Specifying individual properties for every control can become onerous and error prone. For example, suppose a form contains five buttons and each button specifies a width of 150 pixels. If you want to change the width from 150 pixels to 155 pixels, you must make the change five times. *Styles* allow you to set properties on multiple controls all at once. Styles are declared inside the `Resources` property of an XAML element. The following snippet declares a `Style` with the key `NarrowButton` that applies to `Button` controls:

```
<Grid.Resources>
    <Style x:Key="NarrowButton" TargetType="Button">
        <Setter Property="Width" Value="150" />
    </Style>
</Grid.Resources>
```

The `NarrowButton` `Style` will set a `Button`'s `Width` property to 150 pixels. Styles are explicitly set on controls using the `StaticResource` markup extension:

```
<Button Content="Save" Click="Button_Click"
    Style="{StaticResource NarrowButton}" />
```

If you look back at listing B.1, you'll see two other examples of setting a `Style` with the `StaticResource` markup extension. The `Styles` named `PhoneTextNormalStyle` and `PhoneTextTitle1Style` are two of a number of different styles provided by the Windows Phone SDK to allow you easily build Metro-themed applications. You can read more about the built-in styles in chapter 11.

Styles can also be automatically applied to a set of controls. These automatic styles are named *implicit styles*, and are activated by creating a style without a key.

B.4 Binding controls to model objects

One of the more powerful aspects of Silverlight is its ability to separate user interface markup from code logic. Data binding is one of the underlying features that enable UI separation. In the `Button_Click` method we discussed earlier, the code behind needed to know that a `TextBox` control named `nameTextBlock` exists in the UI markup. This knowledge links the UI markup to code behind. If the `TextBox` is renamed, or if another type of control is used instead of a `TextBox`, the code will have to change as well.

Data binding enables you to write code behind that's unaware of the names and types of input controls used in the user interface. Let's say you have a plain C# domain model object with a `UserName` property:

```
public class SampleModel
{
    public string UserName { get; set; }
}
```

In the page contructor, you create a new instance of the model object and assign it to the page's `DataContext` property:

```
DataContext = new SampleModel();
```

The `DataContext` property is used by the Silverlight binding system as the data source when resolving data binding requests made in XAML markup. A data binding request is declared with the `Binding` markup extension:

```
<TextBox x:Name="nameTextBox" Text="{Binding UserName, Mode=TwoWay}" />
```

In the snippet, you declare that the `TextBox` should get its value from a property named `UserName` that exists on the model object referenced by the `DataContext`. Setting the binding `Mode` to `TwoWay` tells the `TextBox` to write any changes back to the `UserName` property as well. Now you can update the `Button_Click` method and replace knowledge of the `TextBox` with code that uses the `DataContext`:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    SampleModel model = (SampleModel)DataContext;
    MessageBox.Show("You entered : " + model.UserName);
}
```

Using plain C# objects works well if data binding is only writing to the model object. If portions of your user interface read from the model object as well, you can help out the data binding system by implementing property change notifications.

B.5 Property change notifications

When a model object is used as a binding source, the binding system checks whether the model object implements the `INotifyPropertyChanged` interface. The `INotifyPropertyChanged` interface declares a single member, an event named `PropertyChanged`. The binding system subscribes to the `PropertyChanged` event and tells the user interface controls to update the values they're displaying when the event is raised. A sample model object that implements `INotifyPropertyChanged` is shown in the next listing.

Listing B.2 Implementing `INotifyPropertyChanged`

```
public class SampleModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    string userName;
    public string UserName
    {
        get { return userName; }
        set
        {
            userName = value;
            if( PropertyChanged != null)
                PropertyChanged(this,
                    new PropertyChangedEventArgs("UserName"));
        }
    }

}
```

① **Define event handler**

② **Raise event**

To implement `INotifyPropertyChanged`, an event named `PropertyChanged` ① is added to the class. The event should be raised whenever a property is changed. Inside the property setter, you determine whether any listeners are subscribed to the event by checking whether the `PropertyChanged` event is null. If the event isn't null, you raise the event and send a new instance of the `PropertyChangedEventArgs` class ②. The name of the changed property is specified when constructing the event args.

In addition to binding user interface controls to model objects, the data binding system also supports binding the property of one control to the property of another control. Binding one control to another is called *element-to-element binding*.

B.6 Element-to-element binding

When designing user interfaces, you often come across usability features that have little to do with the business logic implemented in a model object. For example, a button might be disabled or hidden if the value in a `TextBox` is empty. The model object shouldn't care whether a button is enabled. Let's see how to use element-to-element binding to echo the value entered into a `TextBox` in another control, in this case a `TextBlock`:

```
<TextBlock Text="{Binding Text, ElementName=nameTextBox,
    StringFormat='You entered: \{0\}'}" />
```

You use the `Binding` markup extension to bind the `Text` property of the `TextBlock` to the `Text` property of a `TextBox`. The `ElementName` attribute identifies which `TextBox` to use, in this example the `TextBox` named `nameTextBox`. Finally, you use the `StringFormat` markup extension so that the value isn't merely echoed in the `TextBlock`, but is a formatted message.

Let's return to the example of disabling a `Button` when a `TextBox` is empty. There's no automatic way to bind a `Button`'s Boolean `IsEnabled` property to the length of the `TextBox`'s `Text` string. What you need is a way to convert data during data binding.

B.7 Converting data during data binding

Data binding connects the property in a source object with a property in a target object. Sometimes the source property's type doesn't match the target property's type. Silverlight's data binding system knows how to automatically convert certain types of data. For example, the data binding system will automatically convert the string entered in a `TextBox` to an integer property in a bound object. Silverlight provides a mechanism named *value converters* as a way to convert data types that can't be automatically converted.

At the heart of the value conversion process is the `IValueConverter` interface. `IValueConverter` defines two methods named `Convert` and `ConvertBack`. The `Convert` method is called when copying the source property into the target property. The `ConvertBack` method is used in `TwoWay` binding when the target property's value is copied back to the source property. A sample value converter, named `StringLengthToBooleanConverter`, transforms a string into a Boolean value, based on the length of the string. If the is string empty, the converter returns false:

```
public object Convert(object value, Type targetType,
    object parameter, CultureInfo culture)
{
    string text = value as string;
    return string.IsNullOrEmpty(text) ? false : true;
}
```

To use a value converter in a XAML document, you first declare an instance of the converter as a resource:

```
<Grid.Resources>
    <primer:StringLengthToBooleanConverter x:Key="stringLengthConverter" />
</Grid.Resources>
```

Value converters are specified using the `Converter` attribute of the `Binding` markup extension:

```
<Button Content="Save" Click="Button_Click"
    Style="{StaticResource NarrowButton}"
    IsEnabled="{Binding Text, ElementName=nameTextBox,
➥  Converter={StaticResource stringLengthConverter}}" />
```

We've just scratched the surface of the features built into the data binding system and how to use data binding to separate user interface code from business logic. `DataTemplates` are another feature of Silverlight that enables the designer/developer workflow.

B.8 Using templates to build data model UI

`DataTemplates` allow application code to manage domain model objects without regard to the user interface used to display them. `ListBoxes`, `ContentControls`, and other content-based controls are designed to display generic objects. By default, content-based controls will call the `ToString` method of its contained object, and display the result in the user interface. The default presentation can be replaced using a `DataTemplate`. The following listing shows how to create a `DataTemplate` for a `ContentControl` bound to a sample model object.

Listing B.3 Creating a `DataTemplate` for a `ContentControl`

```
<ContentControl Content="{Binding}" HorizontalContentAlignment="Stretch">
    <ContentControl.ContentTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Polygon Width="50" Height="60" Points="10,0 50,25 10,50"
                    Fill="{StaticResource PhoneAccentBrush}"/>
                <StackPanel>
                    <TextBlock Text="User Name:"
                      Style="{StaticResource PhoneTextGroupHeaderStyle}" />
                    <TextBlock Text="{Binding UserName}"
                      Style="{StaticResource PhoneTextLargeStyle}" />
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </ContentControl.ContentTemplate>
</ContentControl>
```

① **Declare content template**

② **Bind to model property**

The `ContentControl`'s `Content` property is bound to the page's `DataContext` using an empty `Binding` expression. Next, you set the `ContentTemplate` property of the `ContentControl` ① by declaring a `DataTemplate`. The `DataTemplate` uses nested `StackPanels` to display a triangle, a label, and the value of the model's `UserName` property ②.

Though we haven't covered them here, templates are also used to customize the look and feel of controls. A designer can use templates to completely change the way a `Button`, `TextBox`, or other `Control`-derived class appears in the user interface.

B.9 Summary

Silverlight is a powerful application development platform and we've just scratched the surface in this appendix. Silverlight has built-in support for transforms, animations, commands, custom controls, and more. Many Silverlight developers have adopted the Model-View-ViewModel (MVVM) pattern that leverages Silverlight's features to further separate user interface code from business logic.

We take a deeper look at some of the controls in chapter 11. Otherwise there are plenty of good references for learning Silverlight:

- *Silverlight 5 in Action* by Pete Brown, published by Manning Publications Co
- Silverlight.net, a Microsoft portal with links to tools, tutorials, and forums
- "Microsoft Domain-Specific Languages and XAML Specifications," an MSDN page listing the published XAML specifications (see http://mng.bz/rede)

appendix C
AppHub and Marketplace

AppHub and Marketplace are where you go to submit and sell games. Once you've built and tested your Windows Phone application, you want to get your application into the hands of users and start earning some money. The AppHub is where you go to submit your application to the Marketplace. The Marketplace is the only place a user can install your application.

The Marketplace was built to provide Windows Phone users with a single source for downloading and purchasing applications. Microsoft wanted to make it easy for developers to connect with users, get paid for their applications, and license their applications. Microsoft also wanted Windows Phone users to feel safe and secure, knowing that the applications they download have been tested and are free of malware.

The steps necessary for publishing an application are register, build, and submit. Once submitted, Microsoft will put the application through a series of certification tests. If your application doesn't pass certification, it won't be made available to Windows Phone users. Fortunately, the Windows Phone developer tools come with a helpful tool that walks you through automated and manual certification tests. You can track the certification status in your AppHub Dashboard.

The AppHub Dashboard is where you go to review the progress of your application. The Dashboard shows the certification status. Once your application passes certification and is made available to users, you can track downloads, money earned, and user reviews.

At this point, you've probably already registered, but if not, head over to the AppHub, http://create.msdn.com, and register for a new account.

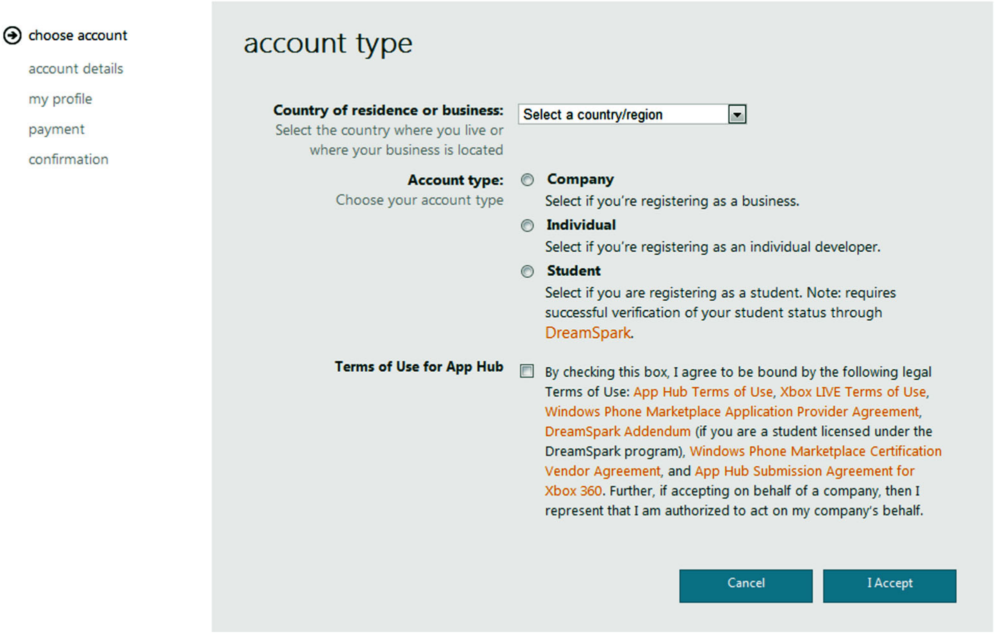
C.1 Registering

Registering for the AppHub isn't as simple as registering for a normal website. Because the AppHub is responsible for collecting money on your behalf, they have a legal obligation to validate that a seller is a real person and/or business. Microsoft will only register individuals and businesses that reside in the countries where they've established the procedures and processes to ensure they follow local tax laws.

> **NOTE** For legal reasons, developers in certain regions around the world aren't allowed to register with the AppHub or submit applications to the Marketplace. In these regions, developers can use a *global publisher* to submit applications on their behalf.

During the registration, you'll be asked for personal and business information. The type of information you'll need to provide will vary based on the account type you choose. There are three different account types: Individual, Business, and Student, shown in figure C.1. Choose Individual if application sales will be reported as part of your personal income. Choose Student if you participate in Microsoft's DreamSpark program. AppHub registration is free for DreamSpark participants. When choosing Business, you'll be asked for the contact information of an individual within your organization who will provide the necessary documentation for identity validation.

Registration isn't instantaneous. Microsoft uses GeoTrust for identity validation and the validation process will take approximately 10 days to complete. After completing registration on the AppHub's website, you'll be contacted by GeoTrust if any additional documentation is required. GeoTrust may request a business's articles of incorporation or charter documents. When the identity validation process is complete, you'll be able to submit applications to the marketplace.



Country of residence or business: Select the country where you live or where your business is located

Select a country/region

Account type: Choose your account type

○ **Company**
Select if you're registering as a business.

○ **Individual**
Select if you're registering as an individual developer.

○ **Student**
Select if you are registering as a student. Note: requires successful verification of your student status through DreamSpark.

Terms of Use for App Hub

☐ By checking this box, I agree to be bound by the following legal Terms of Use: App Hub Terms of Use, Xbox LIVE Terms of Use, Windows Phone Marketplace Application Provider Agreement, DreamSpark Addendum (if you are a student licensed under the DreamSpark program), Windows Phone Marketplace Certification Vendor Agreement, and App Hub Submission Agreement for Xbox 360. Further, if accepting on behalf of a company, then I represent that I am authorized to act on my company's behalf.

Figure C.1 Choosing an AppHub account type

C.2 Submission

You've registered, have an AppHub account, and have built and tested a killer Windows Phone application. Now you need to submit the application to the Marketplace. Where do you start? Applications are submitted from the AppHub website. Once you've logged into your AppHub account, look for the Submit for Windows Phone link on the home page of the AppHub.

The application submission and certification process isn't instantaneous. Each application is run through a series of tests to ensure the application meets the requirements specified in the *Application Certification Requirements for Windows Phone*. The certification process takes approximately five business days to complete.

> **NOTE** If your application uses private or account-restricted web services, you might be asked to provide special access or private builds of the web services to the testing and certification team.

Your application will be rejected if it doesn't meet all of the certification requirements. Microsoft has provided the Marketplace Test Kit as part of the Windows Phone Developer Tools to help developers build applications that pass marketplace certification.

C.2.1 Using the Marketplace Test Kit

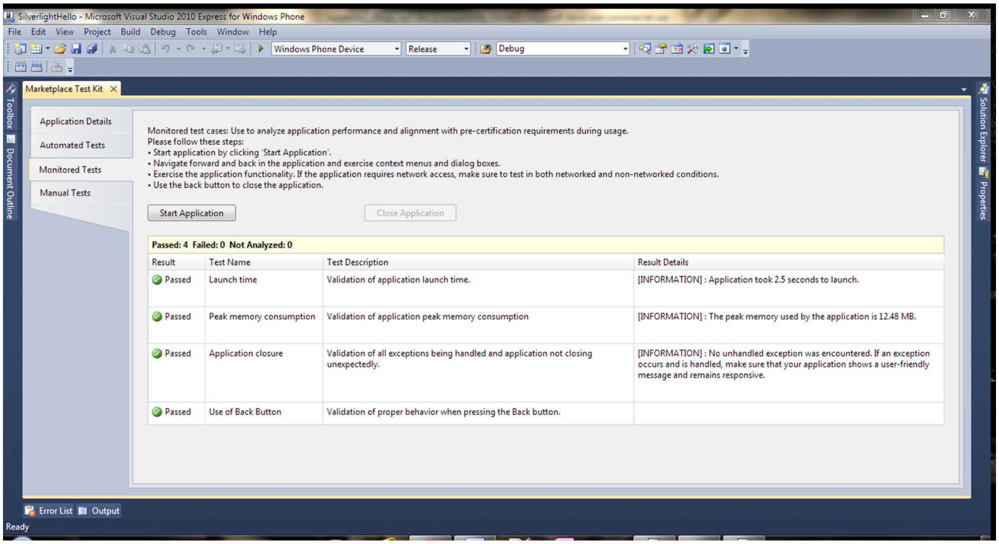
The Marketplace Test Kit includes a series of automated, monitored, and manual tests you can use to ensure your application meets the *Application Certification Requirements for Windows Phone*. The Marketplace Test Kit also helps you assemble the graphics and screenshots that will be submitted along with the application's .xap file. The Marketplace Test Kit is installed when you install the Windows Phone Developer Tools.

The Application Details page of the Marketplace Test Kit prompts you for the location of the application's .xap file, screenshots, and three different-sized tile images. The images are copied into a new folder named SubmissionInfo, which is created under your project's root folder. You'll use the images copied to the SubmissionInfo folder when you submit the application to the Marketplace.

The Automated Tests page of the Marketplace Test Kit inspects the .xap file and the images in the SubmissionInfo folder and checks them against Marketplace requirements. The Monitored Tests page conducts several tests while eavesdropping on your application as it runs on a device. Figure C.2 shows the monitored test results for the Hello World application built in chapter 2.

The Marketplace Test Kit's Manual Tests page contains a checklist of several dozen items that should be checked before submitting the application to the marketplace. Each test provides a drop-down allowing you to check off success or failure of the test. Each test also provides a list of steps or instructions on how to execute the test and why the test is important. Many of the manual tests apply only to applications that implement certain features, such as background file transfers or Music + Videos Hub extensibility.

Once you've passed all the tests, you're ready to submit the application.


SilverlightHello - Microsoft Visual Studio 2010 Express for Windows Phone
File
Edit
View
Project
Build
Debug
Tools
Window
Help
Windows Phone Device
Release
Debug
Toolbox
Document Outline
Marketplace Test Kit
Solution Explorer
Properties
Application Details
Automated Tests
Monitored Tests
Manual Tests
Start Application
Close Application
Error List
Output
Ready

Monitored test cases: Use to analyze application performance and alignment with pre-certification requirements during usage.
Please follow these steps:
- Start application by clicking 'Start Application'.
- Navigate forward and back in the application and exercise context menus and dialog boxes.
- Exercise the application functionality. If the application requires network access, make sure to test in both networked and non-networked conditions.
- Use the back button to close the application.

Passed: 4 Failed: 0 Not Analyzed: 0

Result	Test Name	Test Description	Result Details
✓ Passed	Launch time	Validation of application launch time.	[INFORMATION] : Application took 2.5 seconds to launch.
✓ Passed	Peak memory consumption	Validation of application peak memory consumption	[INFORMATION] : The peak memory used by the application is 12.48 MB.
✓ Passed	Application closure	Validation of all exceptions being handled and application not closing unexpectedly.	[INFORMATION] : No unhandled exception was encountered. If an exception occurs and is handled, make sure that your application shows a user-friendly message and remains responsive.
✓ Passed	Use of Back Button	Validation of proper behavior when pressing the Back button.	

Figure C.2 Viewing test results

C.2.2 Submission checklist

When you submit your application to the Marketplace, you'll be prompted for several different bits of information. The submission process will be easier for you if you have the required information assembled ahead of time. The AppHub provides a list of the required information in the form of a submission checklist. You can view the submission checklist on the AppHub at http://mng.bz/ebia.

To complete the application submission you'll need to provide contact information and a detailed description. You'll also need the tile and screenshot images you selected in the Application Details page of the test kit. You'll need to decide how much to charge for your application (or whether you'll give it away free), and whether you'll provide a trial license. Finally, you'll need to decide on the application category and subcategory. A few of the categories include Games, Business, Sports, and Lifestyle. The full list of categories can be viewed on the Application Submission page in MSDN at http://mng.bz/Rait.

You'll also need to decide how your application will be distributed. You can select from one of three distribution options. Public distribution will publish your application to the Marketplace, where it's visible to every user searching the Marketplace. Targeted distribution is ideal for corporate IT applications because the application isn't listed in the Marketplace and is only available to users who have the exact URL to the application's Marketplace page. Beta distribution allows you to publish the application to a restricted group of users specifically for testing.

C.2.3 Beta testing

The Marketplace's beta distribution program allows you to deliver your application to users either inside or outside your organization who don't have developer-unlocked phones. When submitting a beta application, the application doesn't have to pass the full suite of certification tests. When you submit the beta application, you specify the email addresses of up to 100 beta testers. The email addresses must be linked to the Windows Live ID your users used to set up their phones.

Beta applications are available from the Marketplace for 90 days. After 90 days, the beta license will expire and users will have to uninstall the application.

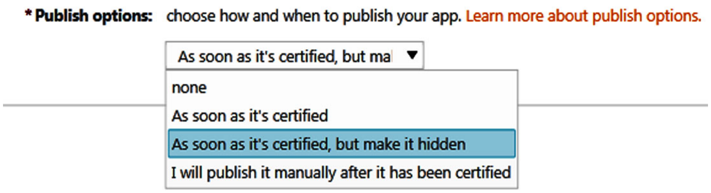
C.2.4 Support for enterprise IT applications

Many companies build private or internal applications that are distributed to their workforce. In the case of mobile applications, the company's IT staff would install these internal applications onto their employees' mobile devices. Windows Phone doesn't support internally deployed applications.

One option for deploying internal applications is for the IT staff to developer-unlock each employee's phone and then use the Windows Phone Developer Tools to side load internal applications onto each phone. The drawback to this option is the cost and overhead for purchasing enough AppHub accounts to unlock every phone. One AppHub account costs $99 and will only unlock five phones, making the cost $20 per phone. The advantage to this option is that internal applications can never be obtained by unauthorized users.

A second option for internal applications is to use the Marketplace's *targeted distribution* feature. An application submitted for targeted distribution goes through the same certification process as a normal application, but the application isn't discoverable by regular Marketplace users. A targeted distribution application is submitted using the normal submission process. When asked for publication options, choose *As soon as it's certified, but make it hidden*, as shown in figure C.3.

Once the application is published to the Marketplace, it'll be hidden from casual users. The application won't appear in any search results, or when browsing application categories. Users are required to have the exact URL of the application in order to view the application details and install the application. Unlike beta distribution, targeted distribution isn't restricted to specified email addresses. If a user has the URL, they'll be able to install the application. You can find the URL in the details section of the application's page on the AppHub.



*** Publish options:** choose how and when to publish your app. Learn more about publish options.

As soon as it's certified, but mal

none
As soon as it's certified
As soon as it's certified, but make it hidden
I will publish it manually after it has been certified

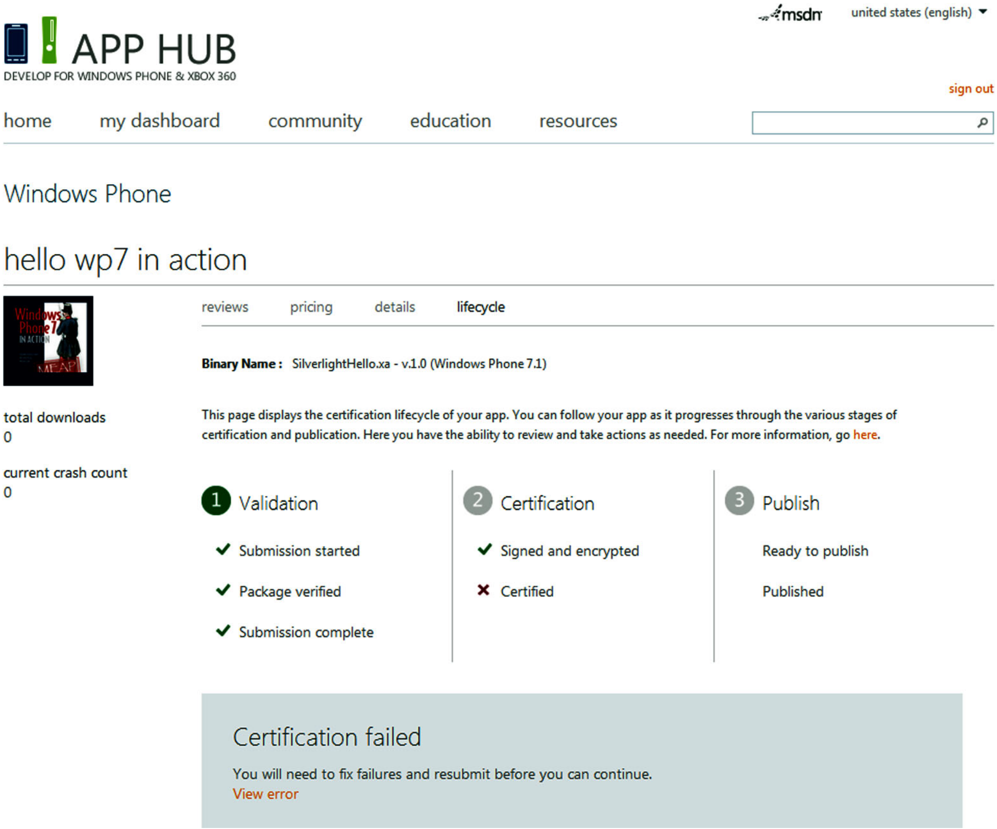
Figure C.3 Publishing an application with targeted distribution

C.3 Certification

Once an application has been submitted, Microsoft will put the application through a series of certification tests. The test and policies are outlined in the *Application Certification Requirements for Windows Phone*, which can be viewed at http://mng.bz/3ZfT. If your application doesn't pass certification, it won't be made available to Windows Phone users. Certification status is found in the application's page in the AppHub. Figure C.4 shows the application page for the Hello World application you built in chapter 2.

The Hello World application isn't a fully functional application and doesn't meet several certification requirements. As you can see in figure C.4, the Hello World application failed the certification process. When an application fails certification, the testing team provides you with a detailed report of the certification polices that weren't met. A portion of the certification failure report for the Hello World application is shown in figure C.5.

Once your application is certified, it'll be ready for publishing. If you choose to publish as soon as it's certified, the application will be available for users to download


APP HUB
DEVELOP FOR WINDOWS PHONE & XBOX 360
msdn
united states (english)
sign out
home
my dashboard
community
education
resources
Windows Phone
reviews
pricing
details
lifecycle
Validation
Certification
Publish

hello wp7 in action

Binary Name : SilverlightHello.xa - v.1.0 (Windows Phone 7.1)

This page displays the certification lifecycle of your app. You can follow your app as it progresses through the various stages of certification and publication. Here you have the ability to review and take actions as needed. For more information, go here.

total downloads
0

current crash count
0

1 Validation
✓ Submission started
✓ Package verified
✓ Submission complete

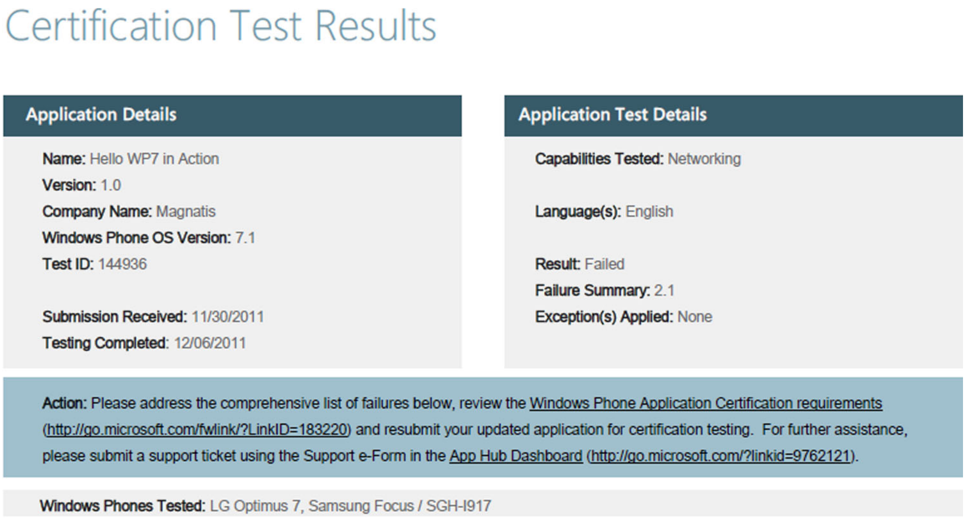
2 Certification
✓ Signed and encrypted
✗ Certified

3 Publish
Ready to publish
Published

Certification failed

You will need to fix failures and resubmit before you can continue.
View error

Figure C.4 The lifecycle section of the application's page in the AppHub



Certification Test Results

Application Details

Name: Hello WP7 in Action
Version: 1.0
Company Name: Magnatis
Windows Phone OS Version: 7.1
Test ID: 144936

Submission Received: 11/30/2011
Testing Completed: 12/06/2011

Application Test Details

Capabilities Tested: Networking

Language(s): English

Result: Failed
Failure Summary: 2.1
Exception(s) Applied: None

Action: Please address the comprehensive list of failures below, review the Windows Phone Application Certification requirements (http://go.microsoft.com/fwlink/?LinkID=183220) and resubmit your updated application for certification testing. For further assistance, please submit a support ticket using the Support e-Form in the App Hub Dashboard (http://go.microsoft.com/?linkid=9762121).

Windows Phones Tested: LG Optimus 7, Samsung Focus / SGH-I917

Figure C.5 A certification failure report

and install. If you chose to publish it manually, you'll need to visit the AppHub when you're ready to make the application available.

The application's page in the AppHub is also where you go to view information about your application such as the number of downloads and number of purchases. If you charge money for your application, you'll also see details about how much money you've earned.

Your application won't sell itself (unless you're really lucky). There are tens of thousands of applications in the Marketplace—how is yours going to get noticed? Promote your Windows Phone application with links to the Marketplace. Inside your application, use the Marketplace tasks to provide searches for other applications from your company. Create a website to market your application that includes links to your application's page in the Marketplace.

Microsoft may choose to feature your application in the Marketplace. Featured applications might be shown in the Marketplace website, or inside the Marketplace application on the phone.

index

Numerics

3D design software 377
3D model 373
3G2 312
3GP 312

A

absolute intensity 214
absolute positioning 432
absolute URI 315
Accelerometer Tool 5, 16, 200, 203, 207, 209, 219, 224
AccelerometerReading 210
AccelerometerReadingEventArgs 209
accent text style 291
AcceptsReturn 298
accounts 110, 114, 118
AcquiringLicense 318
ActivatedEventArgs 61, 69–70, 73
ActivatedTime 69, 75
activation 68, 104
ActiveTiles 248
activity streams 10
ActualHeight 270
ActualWidth 270–271
Add Existing Item 147
Add New Item 269, 277
Add Reference dialog 86
Add Service Reference 356
AddColumn 144
AddMinutes 82
address book 15, 99
AddressChooserTask 15, 104, 109, 366–367
AddressFamily 252
addressing scheme 252
AddressKind 114
AddressResult 109, 367
Advanced Options menu 38
advanced properties 291
AgentExitReason 90
AgentStatus 88–89
airplane mode 96, 229, 232
alarms 58, 74, 77, 80–81, 88
Albums 172
altitude 342, 346
amplitude 333
Android 5, 17–19
angle measurement 350
animation 288, 423
App Connect 171, 176
App Store 5
App.xaml 34
AppConstructedTime 62
appdata 107, 121, 147, 364
AppHub 11, 20, 22, 438, 440–442
Apple 5, 15
application bar 235, 239, 248, 251, 264
application category 441
Application Certification Requirements, for Windows Phone 49–50, 65, 75, 94, 401, 440, 443
application installation 438
application list 8, 33, 50–51, 59, 65, 69
application manifest file 18, 26
application submission 440
application tiles. *See* tiles
Application_Activated 61, 69, 71
Application_Closing 61
Application_Deactivated 61, 67, 71
Application_Launching 61, 65, 87–89, 91, 316
ApplicationBar 260–268
ApplicationBarIconButton 44, 78, 124, 184, 222, 251, 267, 347
ApplicationBarMenuItem 152, 154, 160, 168, 175, 261, 267, 313
ApplicationBarMode 264
ApplicationBarStateChangedEventArgs 267
ApplicationIcon.png 32, 51
ApplicationId 357
ApplicationIdleDetectionMode 75
ApplicationLifetimeObjects 61, 374
applications 101–102
ApplicationSettings 127, 273, 277
apply resource 38, 291
appointments 27, 110, 115–116, 118
AppointmentsSearchEventArgs 117
apps menu 178
Apps.Games 33
Apps.Normal 33

ARM CPU 6
articles of incorporation 439
artwork 50, 191
aspect ratio 380
AssemblyInfo.cs 32
Assets tool 424, 426, 429
assisted GPS. *See* global positioning system
asynchronous pattern 112, 115–116, 242
AsyncState 242
attached properties 37, 305, 308, 429
AttitudeReading 220, 224–225
audio agent 194
audio file 82, 107
audio format 189
AudioMediaStreamSource 330
AudioPlayerAgent 194
AudioStreamingAgent 194
AudioTrack 195
augmented reality application 160, 199
Autodesk Maya 377
automated tests 440
automatic correction 299
AutoPlay 317
AvailableResolutions 161
AvailableTracks 339
AVAudioRecorder 17

B

back buffer 397
Back button 59, 64, 73, 399, 401
back light 379
BackBackgroundImage 245, 248
BackContent 245, 248
background agent 85–91, 194
background audio 31
background image 233, 246
background layer 268
background processing 77, 85
background task 58, 85
background thread 232
Background.png 32, 51
BackgroundAgent 86
BackgroundAudioPlayer 194, 196
BackgroundColor 266
BackgroundImage 49, 245, 248
BackgroundServiceAgent 86
BackKeyPress 48
BackTitle 245, 248
BarFill 204, 207
BasicEffect 380
batteries 37, 75, 85, 389
BeginGetRequestStream 241
BeginGetResponse 242
BeginInvoke 161, 163, 202, 217, 223, 232, 238, 242
BeginTime 79, 81, 83
behaviors 423, 429
beta distribution 441
BinaryWriter 188
binding 434, 436
BindingExpression 140
BindToShellTile 236
BindToShellToast 236
Bing Maps 18, 96, 342
Bing Query Language 103
Bing Search application 103
BingMapsDirectionsTask 96, 342, 344–345
BingMapsTask 96, 342–344
bit fields 286
BitmapImage 295
bitwise operation 289
Blender 377
Bluetooth 96
BorderBrush 292
border-color 364
border-style 364
BorderThickness 166, 292
bounding circle 266
BoundingBox 393
BoundingSphere 384, 386, 391–392
breakpoints 33
broadcast receiver 18
browser viewport 362
BufferDuration 185
buffering 318
BufferingProgress 317
BufferProgress 194
BufferReady 185
Build Action 207, 274–275, 315, 347
BuildNavigationUri 47
built-in applications 94, 96, 172
business accounts 439
business logic 430
ButtonState 412–413
Buy Now button 101
byte array 242

C

calendar 110
CalibrationEventArgs 217
callback method 241
camera application 6, 16, 104, 154, 163
camera button 58, 160, 163
camera hardware 159, 165
camera position 380, 383
camera roll 152, 173
CameraButtons 159, 163
CameraCaptureTask 16, 104, 149, 154
CameraOperationCompletedEventArgs 158, 164
CameraType 159
CameraVideoBrushExtensions 162
CancelEventArgs 48, 402
CanGoBack 394
CanSeek 321–322, 326, 332
Canvas 287, 395, 426, 432
capabilities 26
Capability Validation Test 26
capacitive touch 6
CaptureCompleted 163
CaptureImageAvailable 163
CaptureStarted 163
CaptureThumbnailAvailable 163
CC 99
CDMA cellular network 231
cellular network 228–229, 342
cellular radio 346
CellularMobileOperator 229
center point 409, 413, 420
certification guidelines 8
certification process 440
certification tests 438, 443
channel name 236
channel URI 234–235, 239, 244, 248
ChannelUriUpdated 237
CharacteristicUpdate 231
chat application 250
CheckBox 76, 265, 297
ChosenPhoto 154–155
CivicAddress 114
class library 31, 373
CleanUpCamera 161, 164
CLHeading 16
ClickMode 296
clipboard 42, 44, 239
clipping plane 380–381

closing sockets 254
ClosingEventArgs 61
CLR. *See* Common Language Runtime
code-behind 45, 268, 430–431, 434
codec 312
CodecPrivateData 322, 332
CodePlex 301, 308
collision detection 391
ColumnDefinitions 432
ColumnSpan 432
command bar 22, 287
Common Language Runtime 17
Compaq iPaq 4
CompassReading 203, 214
CompiledQuery 142
CompleteName 114
CompositeTransform 167
conditional compilation 52
cones 377
ConnectAsync 253
connected client 250
connection string 135
connection-less 249
ConnectionSettingsTask 96
ConnectionStatusChanged 237
ConnectivityManager 19
ConsumerID 177
contact records 111
ContactAddress 114
ContactCompanyInformation 114
ContactEmailAddress 114
ContactPhoneNumber 114
contacts 9, 27, 110–111, 113
contacts application. *See* People Hub
contacts database 9, 15, 108, 111
ContactsSearchEventArgs 113
ContainmentType 393
ContainsKey 193, 278
content pipeline 377
content project 373, 377
content provider 18
ContentControl 114, 116, 118, 303, 354, 436
ContentIdentifier 102
ContentManager 377, 379, 385, 409, 413, 420
ContentPanel 36–37, 166, 206, 251, 312, 352, 432
ContentReadyEventArgs 163
ContentTemplate 112, 354, 437
ContentType 101–103
ContextMenu 301, 304
ContextMenuService 304
continuous connection 249
contrast text style 291
control playback 313
control template 41
ControlStoryboardAction 429
ConvertBack 436
cookies 241
coordinate system 201, 220, 224, 380
CopyAbsoluteBoneTransformsTo 379
copying, to clipboard 42
Core Motion 16
CoreAnimation 15
CornerRadius 292
CreateDatabase 144, 147
CreateDatabaseSchemaUpdater 144, 146
CreateDirectory 129, 131, 137, 365
CreateFile 129, 131, 168, 365
CreateFromAxisAngle 384–385
CreateInstance 189
CreateLocationRect 353
CreateLookAt 381, 384
CreatePerspectiveFieldOfView 381
CreateScale 381–382
CreateWorld 381–383, 386
creating reviews 101
CredentialsProvider 355
CRUD 135, 137
CSS 363
CSS3 358
cubes 377
current location 343
CurrentRegion 196
CurrentState 317–318, 337
CurrentStateChanged 318, 337
CurrentValue 200, 202, 209, 215, 218, 222, 418
CurrentValueChanged 200, 202, 207, 222–223, 418

D

daily recurrence 81
Dalvik 17
dark theme 266, 274, 290–291
dashboard 438
data binding 45, 115
data block 249
data context 135
data roaming 229
data stream 249
DatabaseExists 137
databases 16, 18, 133–134, 143
DatabaseSchemaUpdater 144–145
DatabaseSchemaVersion 144, 146
DataContext 64, 68–69, 135, 138, 140–141, 434
datasource 136
DataTemplate 77, 80, 112, 119, 123, 280, 303, 436
DatePicker 301, 303
DateTime 63, 68–69, 75, 79, 117, 123, 303
DateTimeOffset 200
DeactivatedEventArgs 61
DeactivatedTime 67, 70
deactivation 60, 95, 104
dead zone 408
debug target 33
debugging 33, 46, 91
DecodeImage 158, 164
DecodeJpeg 154, 158, 169
deep link navigation 171
_default task 34, 177
DefaultItem 272
DefaultMaximumItems 117
DefaultTask 180, 193
degrees 350
DeleteAllOnSubmit 139
DeleteDirectory 129, 132
DeleteFile 129, 132
DeleteOnSubmit 139
Delta 416–417
Delta2 416
departure location 345
dependency property 204
DependencyObject 263
descending 138, 141
design language 8
DesiredAccuracy 346
destination location 343, 345
developer key 355, 357
Developer Registration Tool 22, 33
developer unlock 442
developer unlocked phones 22
Device tool 424
DeviceAcceleration 220
DeviceExtendedProperties 27
DeviceNetworkInformation 229
DeviceRotationRate 220
DeviceType.Emulator 33

device-width 362
digital rights management 107, 318, 322
DirectoryExists 129–131, 137
DirectoryInfo 122
DirectX 13, 21, 372, 380
DispatchTimer 183, 209, 222
displaying filtered data 276
displaying name searches 113
DisplayName 97–98, 107, 112, 114, 358
distribution options 441
DOCTYPE 362
document centric applications 14
Documents folder 16
dodecahedron 377
domain model 430, 434
dormant application 14, 57, 60
dormant state 60, 65–66, 68, 75, 165
DoubleAnimation 288
DoubleTap 41–43, 307, 415
DownloadProgress 317
DragComplete 307, 415–417
DragCompletedGestureEventArgs 307
DragDelta 307
DragDeltaGestureEventArgs 307
DragStarted 307
DragStartedGestureEventArgs 307
drawing primitive 38
DrawModel 379–380
DrawString 394
DreamSpark 11, 439
driving directions 342
DRMHeader 321

E

ECMAScript 5 358
edges 377
ElapsedTime 376
element to element binding 435
ElementName 314, 320, 435
Ellipse 38, 293, 354, 426
email address 106, 114
email searches 113
EmailAddress 112
EmailAddressChooserTask 15, 104, 109
EmailAddressKind 114
EmailComposeTask 15, 96, 98
EmailNameOrAddress 299
EmailResult 109
EmailSmtpAddress 299
embedding images 295
emulator 21, 33, 97, 152, 178, 210, 255, 312
EnabledGestures 415–416
EnabledPlayerControls 194
encoding 242, 253
EndDate 117
EndGetRequestStream 242
endpoint 249
EndTime 118
EndTimeInclusive 117
enter key 300
EntityType 358
equator 213, 350
ErrorOccurred 237
establishing connections 253
ethernet 231
EventArgs 74, 217, 262, 306
ExceptionRoutedEventArgs 319
Exchange Image File Format 155
ExecutionTimeExceeded 90
ExifLib 155
ExifOrientation 157
ExifReader 157
ExpirationTime 79, 81, 87–88, 91
Expression Blend 4, 15, 20, 38, 262, 423
ExtendedTask 86
extending Picture Hub 176
extending Picture Viewer 178
Extensible Application Markup Language. *See* XAML
extension methods 114
extension points 181
ExtensionName 177
extensions 10, 167

F

Facebook 9, 96, 110, 173, 176
fast application switching 4, 14, 165, 201
FastForward 194, 196
field of view 380
file picker 151
file system 14, 16, 18, 129
FileExists 129–130, 187, 316
FileId 181
FileInfo 122
FileMode 130
FileStream 122, 129
fill light 379
FillFrame 325
filter type 112
FilterKind 112–113
FindName 263
FirstOrDefault 114, 118, 175
first-time initialization 65
Flick 41–42, 307, 415, 421
FlickGestureEventArgs 307
FMRadio 6, 27, 196
focusing camera 163
font-family 363
FontSize 41, 275
footers 362
foreground application 57, 274, 302
ForegroundColor 266
FormatCoordinate 350
FormatException 108
FormatPayload 255
frame of reference 224
frame rate 375
FrameReported 306
frames per second 338, 376
FrameworkDispatcher 183, 185
FrameworkElement 263, 293
free fall 208, 212
FreeDrag 415
frequency 196–197, 331
friction 209, 212
FromMilliseconds 183, 209, 215, 218, 222
FromSeconds 91
FromStream 189
FrontFacing 159
front-facing camera 151, 159
FTP 17
full press 163

G

game assets 373
game camera 376, 380
game controller 407, 411
game loop 375–376, 385, 389
game pad 407, 411
GamePlayComponent 376, 378, 404
GamerServices 27
Games Hub 9, 33, 50–51
GameTimer 374–376, 389
GameTimerEventArgs 376, 390
garbage collection 14
generic class 200, 202

Geocode Service 355
GeocodeLocation 357
GeocodeResponse 358
GeocodeResult 358
GeocodeServiceClient 356–357
geocoding 355
GeoCoordinate 345–346, 349, 353–354
GeoCoordinateWatcher 11, 346–348, 351
geographic coordinate 345
geographic north 214
geometric shapes 377
GeoPositionAccuracy 346, 348
GeoPositionChangedEventArgs 349
GeoPositionPermission 349
GeoPositionStatus 349
GeoPositionStatusChanged-EventArgs 349
GeoTrust 439
Gesture API 415
gesture processing 415
GestureBegin 307
GestureCompleted 307
GestureEventArgs 43, 307
GestureInput 416
GestureListener 306–307
gestures 415
GestureSample 415, 417
GestureService 307
GestureType 415–416
GetActions 79, 87
GetAngleFromExif 156, 159
GetBytes 242, 253
GetCapabilities 407
GetData 186
GetDiagnosticAsync 323
GetDirectoryNames 129
GetDistanceTo 350
getElementById 367
GetFileNames 129, 188
GetImage 175
GetIsNetworkAvailable 231
GetLastWriteTime 188
GetPosition 306
GetPreviewBufferArgb32 160
GetPreviewBufferY 160
GetPreviewBufferYCbCr 160
GetResourceStream 191, 317
GetResponseStream 243
GetSampleAsync 323, 327, 331–332
GetState 407, 409, 412
GetThumbnail 175
GetUserStoreForApplication 129–130, 137, 168, 187–188, 316, 365
global positioning system 5–6, 11, 346
global publisher 439
globalization 8
GoBack 45–46, 48, 394
Google 5, 17
GoToState 288–289
GPS receiver 342, 346
GPS. *See* global positioning system
GPU. *See* graphics processing unit
GradientBrush 426
Graphics Device Interface 13
graphics processing unit 372, 374, 380
GraphicsDevice 374, 379, 394, 397
gravity 208, 211, 220, 224, 419–420
group header style 291
GroupName 297
GSM cellular network 231
guaranteed delivery 249
Guid 88
Gyroscope 4, 16, 200, 203, 207, 217, 219
GyroscopeReading 218–219

H

half press 163
hardware acceleration 13, 358
hardware back key 48
hardware buttons 6
hardware input panel 4
hardware keyboard 22, 41
hardware specifications 6, 17, 196
HasValue 297
header 302–303, 305, 362
HeaderTemplate 303, 306
HeadingAccuracy 215–216
hello world 30–50
Hewlett-Packard Jornada 4
hidden applications 442
High Level Shader Language 380
HighScore 123
HighScoreDatabaseRepository 136, 139, 143–144
HighScoreFileRepository 129
HighScoreSettingsRepository 127
HLSL. *See* High Level Shader Language
horizontal intensity 214, 216
HorizontalAccuracy 346
HorizontalDrag 415–416
HorizontalScrollBarVisibility 298
host application 90
Hover 296
HTML 430
HTML 5 342, 358, 362, 368
HTTP 10, 17, 19, 228, 240, 335
HTTP POST 234, 239
HttpNotificationChannel 236
HttpNotificationReceived 238
HttpWebRequest 10, 17, 19, 228, 240, 242, 249
HttpWebResponse 10, 17, 19
Hub 8
HubType 190
HyperlinkButton 94, 296

I

IAsyncResult 242
ICommand 45
icon selector 262
icons 207, 222, 263
IconUri 44, 261, 267
ID_CAP_APPOINTMENTS 27, 116
ID_CAP_CONTACTS 27, 111
ID_CAP_GAMERSERVICES 27
ID_CAP_IDENTITY_DEVICE 27
ID_CAP_IDENTITY_USER 27
ID_CAP_ISV_CAMERA 27, 159
ID_CAP_LOCATION 27, 346
ID_CAP_MEDIALIB 27
ID_CAP_MICROPHONE 27
ID_CAP_NETWORKING 27
ID_CAP_PHONEDIALER 27, 97
ID_CAP_PUSH_NOTIFICATION 27
ID_CAP_SENSORS 27, 201
ID_CAP_WEBBROWSER-COMPONENT 27
ID_HW_FRONTCAMERA 27, 159
identity matrix 225
identity validation 439
IDictionary 193

idle detection mode 59, 75
idle timeout 58, 75
IdleDetectionMode 75, 419
IEnumerable 114, 118
IHighScoreRepository 126, 129, 136, 138, 141
IIS Smooth Streaming Client 335
IList 264, 266
image capture 163
image decoding 164
image design 266
image editing 150, 165
image picker 294
image processing 160
image rotation 155
image stream 163
ImageBrush 151, 154, 162, 273
Images folder 78
ImageSource 274
implicit styles 433
indeterminate 293, 297, 302
individual accounts 439
individualizing 318
ingestion tool. *See* Marketplace Test Kit
InitializeApplicationBar 263
InitializeCamera 161
InitializeComponents 263
innerText 367
INotifyPropertyChanged 45, 141, 434
input control 41
InputScope 41, 246, 299–300
InputScopeNameValue 299–300
InsertAllOnSubmit 138
InsertOnSubmit 138
installation folder 122, 146
installing custom ringtones 107
integrating maps 342
intents 18
interaction model 15
InterfaceConnected 231
InterfaceDisconnected 231
internal applications 442
Internet Explorer 22, 96, 342, 358–359
Internet Information Server 335
InterNetwork 252
inter-process communication 14
InvalidOperationException 201, 344–345, 417
InvokeScript 366–367
IOrderedQueryable 142
iOS 14–17
IP address 250
IPEndPoint 252
iPhone 5, 7, 16
iPod 16
IQueryable 142
IsAllDayEvent 118
IsApplicationInstancePreserved 70, 73
IsCameraTypeSupported 159
IsCellularDataEnabled 229
IsCellularDataRoamingEnabled 229
IsChecked 265, 281, 297–298, 302
IsDataValid 201, 203
IsEnabled 79, 89, 261, 306, 402–403, 418, 436
ISensorReading 200
ISETool. *See* Isolated Storage Explorer tool
IsGestureAvailable 417
IsIndeterminate 293
IsLocked 76
IsMenuEnabled 265
IsMenuVisible 267
IsMuted 319
IsNavigationInitiator 67, 72
IsNetworkAvailable 230
IsNullOrEmpty 247
isolated storage 14, 16, 18, 107, 361, 364
Isolated Storage Explorer tool 25, 147
IsolatedStorage 121
IsolatedStorageFile 122, 129–131, 187–188, 316, 365
IsolatedStorageFileStream 130–131, 168, 174, 315–317
IsolatedStorageSettings 16, 18, 122, 127, 272, 277
isostore schema 107, 121, 136
IsoStorePath 107
IsPinnedToStart 114
IsPlaying 393
IsPrivate 118
IsRunningSlowly 376
IsScheduled 79, 87, 90
IsScriptEnabled 361
IsSupported 201, 203, 209, 214, 218, 222, 418
IsThreeState 297, 302
IsTrial 52, 102
IsUnknown 346, 353
IsVisible 37, 265
IsWiFiEnabled 230
IsZoomEnabled 305
item template 39
ItemSource 80, 125
ItemsSource 251, 283
ItemTemplate 78, 123, 184, 189, 280
IValueConverter 45, 436

J

JavaScript 361, 365
JavaScript/HTML application 363
JPEG 294
JpegInfo 157

K

key light 379
KeyFrameFlag 323
keypad 5

L

LabeledMapLocation 345
lambda expression 183
landmarks 343
Landscape Flat 211
landscape orientation 260, 264, 285–290
Landscape Standing 211
LandscapeLeft 286
LandscapeRight 286
LargeChange 301, 319
LastExitReason 90
latitude 213, 342, 346, 350, 355, 357
launch URI 243
LaunchedTime 65, 70
launchers 9, 16, 18
LaunchForTest 91
launching 59, 61, 65, 87, 365
LaunchingEventArgs 61
layout control 432
LayoutRoot 269, 277, 288, 395, 402, 431–432
LeftButton 412
LicenseInformation 52, 102
licensing 438
Life Maximizer 3
lifetime events 59, 61
light theme 266, 274, 290–291
lighting algorithm 379
lighting source 379

linear acceleration 221
linefeed 299
LinkedIn 96
LINQ to SQL 16, 19, 122, 132, 135, 141–143
ListBox 77, 123, 184, 186, 188, 251, 280, 436
ListPicker 301
LoadCompleted 360
LoadDeviceInformation 230
LoadedPivotItem 281
LoadingPivotItem 281
LoadJpeg 167, 169
LoadPhoneNetworkInformation 231
local database 16, 18
location service 199, 341, 346
location-aware application 6, 11, 341, 355
LocationRect 353
lock button 75
lock screen 58, 74–75, 419
long term storage 67
longitude 342, 346, 350, 355, 357
LongListSelector 301
look-ahead position 380–381, 383

M

M4A 312
M4V 312
magnetic field 213, 216
magnetic north 214
MagneticHeading 214
MagnetometerReading 214–216
malicious applications 14
malware 438
Managed DirectX 372
ManifestInfo 339
ManifestReady 338
manipulation event 43
ManipulationCompleted 43, 306
ManipulationDelta 43, 306
ManipulationStarted 43, 306
manual tests 440
MapLayer 352
MapPolygon 353
MapPolyline 352–353
Maps 299, 347, 352
MapView 18
margin of error 408
Marketplace Test Kit 25, 440
MarketplaceContentType 101–102
MarketplaceDetailTask 52, 96, 100, 102
MarketplaceHubTask 96, 100
MarketplaceReviewTask 96, 100–101
MarketplaceSearchTask 96, 103
markup extension 291, 433–434, 436
markup language 430
MathHelper 381, 384–385, 419
matrix 200, 220, 379, 381–382
max database size 136
MaxImageSize 192
MaximumTouchCount 407
MaxLength 299
MaxValue 79
MaxWidth 272
media container 187, 189, 311, 335
media library 153
media player 16
MediaElement 17, 19, 334
MediaElementState 317, 319
MediaEnded 317
MediaFailed 319–320, 337
MediaHistory 27, 191–192
MediaHistoryItem 191
MediaLibrary 172, 175–176, 179
MediaOpened 317
MediaPlayer 16, 19
MediaPlayerLauncher 16, 19, 96
MediaRecorder 19
MediaSourceAttributesKeys 321, 326, 331
MediaStream 328
MediaStreamAttributes 322
MediaStreamDescription 322, 324, 326–327, 330–331
MediaStreamSample 323–324, 328, 332
MediaStreamSource 17, 19, 27, 320–334
MediaStreamType 323, 332
megapixel 6
MemoryQuotaExceeded 90
MemoryStream 174, 185, 187, 323, 326, 328, 332
menu. *See* application bar
MenuItem 305
MenuItems 261
mesh 377, 379
MessageBox 48, 217, 242, 262, 319, 355, 358, 433
MessageBoxButton 48, 262, 305
MessageBoxResult 48
metadata 155
MethodAccessExceptions 122
Metro design language 8, 35
MFMailComposeViewController 15
MFMessageComposeViewController 15
Microsoft Developer Network 11
Microsoft Location Service 346
Microsoft Outlook 110
Microsoft Push Notification Service 8, 11, 232, 234, 239
Microsoft SQL Server Compact 16, 19, 122, 132
Microsoft.Devices 27, 33, 162, 392
Microsoft.Devices.Accelermoter 16
Microsoft.Devices.Radio 6, 27, 196
Microsoft.Devices.Sensors 27, 203, 220
Microsoft.Net .NetworkInformation 19
Microsoft.Phone 154
Microsoft.Phone .BackgroundAudio 194
Microsoft.Phone.Controls 27, 269, 360
Microsoft.Phone.Controls.Maps 346, 352
Microsoft.Phone.Data.Linq 133, 144
Microsoft.Phone.Data.Linq .Mapping 133
Microsoft.Phone.Info 27
Microsoft.Phone.Net .NetworkInformation 228, 231
Microsoft.Phone.Notification 27, 236
Microsoft.Phone.Scheduler 78
Microsoft.Phone.Shell 247, 261
Microsoft.Phone.Tasks 27, 95, 98, 104
Microsoft.Phone.UserData 15, 27, 110
Microsoft.System.Devices 200
Microsoft.Xna.Framework 27, 172, 200, 203, 220
Microsoft.Xna.Framework .Audio 17, 19, 27

Microsoft.Xna.Framework
.Graphics 396
Microsoft.Xna.Framework.Input
.Touch 416
Microsoft.Xna.Framework
.Media 16, 19, 174
Microsoft's Push Notification
Service 17
minimum hardware
requirements 6
minutes 350
MinValue 68–69
MobileBroadbandCdma 231
MobileBroadbandGsm 231
Model-View-ViewModel 45, 111,
269, 276
monitored tests 440
monthly recurrence 81
Motion API 418
Motion sensor 4
MotionReading 202, 220, 223,
419
Mouse API 411–412
mouse events 41
MouseEnter 296
MouseLeftButtonDown 296
MouseLeftButtonUp 296
MouseState 412–413
MovementThreshold 348
MP3 107, 312
MP4 312
MPEG-4 335
MSDN. *See* Microsoft Developer
Network
multicast 249
multi-core processor 14
multi-line editing 298
multi-point touch 5–6, 407,
415–416
multitasking 14, 57
Music + Videos Hub 6, 18–19,
171, 181, 192–193, 440
MVVM. *See* Model-View-View-
Model

N

NameOrPhoneNumber 299
NameValue 300
native applications 94, 96, 103,
108
navigatedFromTime 67, 70, 72
navigatedToTime 63
NavigateToString 360
NavigateUri 296
navigation application 15, 34,
45
navigation URI 237
NavigationContext 47, 180, 406
NavigationEventArgs 63, 67,
229, 237
NavigationFailedEventArgs 34
NavigationPage 34, 180
NavigationService 15, 34, 45, 49,
62, 394
NavigationUri 82–83
.NET Compact Framework 4, 14
network bandwidth 22
network communication 228
network connectivity 227
Network Information API 228
network programming 17
network quality 334
NetworkAvailabilityChanged
230
NetworkInterface 19, 22, 229,
231
new item dialog 111, 277
new project dialog 30
NoData 349
notification channel 233–234
NotificationChannelUri-
EventArgs 237
NotificationEventArgs 238
notifications. *See* push
notifications
NotificationType 231
NotifyComplete 87, 90, 196
NotifyEventArgs 367
NotImplementedException 329,
333
NotSupportedException 328
Now Playing 191–192
NSUserDefaults 16
nullable boolean 297
null-coalescing operator 73
Number input scope 42

O

Object Relational Mapping 132
Objective-C 14
Objects and Timeline
panel 262, 425–427, 429
ObjectTrackingEnabled 140
obscuration 58, 74–76, 97
ObscuredEventArgs 74, 76
ObservableCollection 125, 127,
251
octohedron 377
OData API 132
Office Hub 274
OnBackKeyPress 402
OnDraw 374, 376, 396, 403
OnInvoke 86
OnNavigatedFrom 48, 67, 72,
165, 277, 304, 375–376
OnNavigatedTo 63–64, 272,
278, 375–376, 418, 421
OnNavigatingFrom 48, 67
OnPlayStateChanged 196
on-screen keyboard. *See* software
input panel
OnUpdate 376, 389, 395
OnUserAction 195
OpenCORE 19
OpenFile 129–130, 187
OpenFileDialog 151
OpenGL ES 15
OpenMediaAsync 321–322, 324,
326, 331
orderby 138, 141
OrientationChanged 289
OrientationChangedEventArgs
289
OriginalFileName 154
ORM. *See* Object Relational
Mapping
Outlook 110

P

padding 296
padding-bottom 364
padding-left 364
padding-right 364
padding-top 364
page navigation history 45
page stack 46
pageConstructedTime 63
PageOrientation 286, 289
PageTitle 425
Panorama 31, 39, 259, 268, 273–
275
PanoramaItem 268–269, 271–
272
partial address 343
passing parameters 47
paste from clipboard 42
pausing game play 401
PCM data format 187
People Hub 9, 96, 104, 109–110,
268
percentage complete 293
PeriodicTask 85, 87–89, 91

phase angle 333
phone dialer 15, 96–97
phone number 9, 113
phone registration 23
PhoneAccentBrush 40, 292, 302, 307
PhoneAccentColor 38, 427
PhoneApplicationFrame 34–35, 59, 61, 74
PhoneApplicationPage 36, 46, 48, 72, 104, 261, 263, 269
PhoneApplicationService 59, 61–62, 68, 70, 75, 87, 419
PhoneBackgroundBrush 38
PhoneBorderBrush 292, 302
PhoneBorderThickness 292
PhoneCallTask 15, 27, 59, 96–98
PhoneChromeBrush 306
PhoneContrastBackground-Brush 88, 292–293
PhoneFontFamilyNormal 36
PhoneFontSizeLarge 80
PhoneFontSizeMedium 40
PhoneFontSizeMediumLarge 41
PhoneFontSizeNormal 36, 40–41
PhoneForegroundBrush 36, 302
PhoneMargin 40–41, 112, 114
PhoneNumber 97–98, 109, 112
PhoneNumberChooserTask 15, 104, 108–109
PhoneNumberKind 114
PhoneNumberResult 108
PhoneNumbers 115
PhoneStrokeThickness 293, 306
PhoneSubtleBrush 293
PhoneTextAccentStyle 291
PhoneTextContrastStyle 88, 291
PhoneTextExtraLargeStyle 40, 280, 291
PhoneTextGroupHeaderStyle 95, 112, 116, 291
PhoneTextHugeStyle 291
PhoneTextLargeStyle 40, 124, 166, 280, 291
PhoneTextNormalStyle 80, 124, 291, 433
PhoneTextSmallStyle 80, 291
PhoneTextSubtleStyle 81, 124, 280, 291
PhoneTextTitle1Style 124, 291, 433
PhoneTextTitle2Style 291
PhoneTextTitle3Style 291
PhoneTouchTargetOverhang 95, 296
Photo Chooser application 151
PhotoCamera 16, 150–151, 159, 161, 163, 165
PhotoChooserTask 104, 149, 151
PhotoResult 153, 155
Photos_Extra_Hub 177
Photos_Extra_Share 177, 180
Photos_Extra_Viewer 177, 179
physical address 109, 114
picture viewer application 178
PictureAlbum 172
PictureDecoder 154, 159, 168, 176
Pictures Hub 9, 18, 171, 174, 176, 180, 268, 271
PinchComplete 415
PinchCompleted 307
PinchDelta 307
PinchGestureEventArgs 307
PinchStarted 307
PinchStartedGestureEventArgs 307
PinnedToStart 112
PiOver4 381, 419
Pitch 220, 222, 225
Pivot 8, 31, 39, 110, 228, 235, 259
PivotItem 112, 116, 235, 239, 243, 245
PivotItemEventArgs 282
PixelHeight 157, 168
PixelWidth 157, 168
playback position 322
player movement 381
PlayerContext 192–193
PlayerState 194–195
playhead 427
playing audio 189
PlayState 195
PNG 294
Pocket PC 2000 4
port number 252
Portrait Flat 211
portrait layout 210
portrait orientation 37, 260, 264, 285–290
Portrait Standing 211
PortraitDown 286
PortraitOrLandscape 286
PortraitUp 286
PositionChanged 346, 349, 353
power state 196
PowerMode 196–197
PrepareVideo 326
preview buffer 161
preview image 160
PreviewResolution 161
primary key 134, 138
Prime Meridian 350
product id 25, 33, 101, 147
ProgressBar 293
ProgressIndicator 318
progressive download 334
project properties 51–52, 69
project template 31
property change notification 434
property editors 38, 262, 291, 294, 423
property element syntax 261, 275
PropertyChanged 141, 434
PropertyChangedEventArgs 435
ProtocolType 252
public distribution 441
purchasing applications 438
push enabled application 234
push notification channel 236
push notifications 11, 17, 228
Pushpin 352–354

Q

quadrilateral 377
quaternion 200, 220
query expression 141
query string 47, 179, 181, 193
queryable 137
QueryString 47, 180, 193
QueueUserWorkItem 232
quota 122
QWERTY 41

R

RadioButton 112, 116, 246, 248, 277, 297, 405, 418
RadioRegion 196
random number generator 125
random value 279
raw image data 166
raw notification 233, 238
raw touch 41
reactivation 70
ReadAccelerometerData 210
ReadCompassData 215

ReadGesture 306, 415–416
ReadGyroscopeData 218
reading location data 346
ReadingChanged 209
ReadJpeg 157
read-only databases 136, 146
rebuild solution 84, 145
ReceiveAsync 254
recently played 192
Record Keyframe 427
recording audio 185
recurrence patterns 81
RecurrenceType 81, 83
redistribution license 395
refactoring 34
reference database 146
relative motion 346
RelativeTransform 162
reminders 58, 74, 77, 81
renewing background agents 91
ReportGetDiagnosticCompleted 323
ReportGetSampleCompleted 323, 328, 330, 332
ReportGetSampleProgress 323
ReportOpenMediaCompleted 322, 326
ReportSeekCompleted 323, 333
RequestStreamState 241
resolution 161
resource dictionary 42, 280
resource menu 38
resource picker 38, 291
ResourceIntensiveTask 85
response document 242
restoring application state 272, 278
RestrictTracks 340
reusable control 39
reverse geocoding 355–356
ReverseGeocodeAsync 357
ReverseGeocodeCompleted 357
ReverseGeocodeCompleted-EventArgs 357
ReverseGeocodeRequest 357
RGBA 324, 327
rich graphics applications 21
RichTextBox 35, 40
ringtones 104, 107
roaming 229
RootFrame 74
RootFrame_NavigationFailed 34
RootLayout 360
RootPictureAlbum 173
RootVisual 34
RotateBitmap 158
RotateTransform 162
rotation angle 158
rotation matrix 225
rotational velocity 217
RotationMatrix 224
RotationRate 218–219
RoutedEventArgs 318
RowDefinitions 432
RowSpan 432
RTCP protocol 335
RTP protocol 335
RTSP protocol 335
run time data 72
running behind the lock screen 75
running state 60
RuntimeType 33

S

sample pictures 173
sampling rate 331
sandbox 14, 25–26, 119, 121, 129
SaveContactTask 104
SavedPictures 173–175
SaveEmailAddressTask 16, 104, 106
SaveJpeg 167
SavePhoneNumberTask 16, 104–105
SaveRingtoneTask 104, 107
saving email addresses 106
saving phone numbers 105
saving ringtones 107
scalable vector graphics 358, 363
ScaleVisibility 352
Scheduled Action Service 77–91
scheduled actions 58, 78
scheduled task 77
Scheduled Tasks Agent 31
ScheduledAction 79, 81, 85
ScheduledActionService 58, 83–85, 91
ScheduledNotification 81, 83, 87
ScheduledTask 79, 85
ScheduledTaskAgent 85–86
Scheduler API 81
screen coordinates 352
screen resolution 6, 362
ScriptNotify 366–367
ScrollBarVisibility 298
ScrollViewer 245
search button 58
search term 112, 343
SearchAsync 112–113, 116
SearchCompleted 112–113, 116
searching for contacts 111
SearchTask 96, 103
SearchTerm 103, 343–344
secondary tile 245, 247
security capabilities 26
SeekAsync 322, 328, 333
Segoe font 363
SelectedIndex 272, 278
SelectedItem 83, 272
SelectionChanged 272, 281
self-portrait camera 159
SendAsync 255
sensitivity area 408
Sensor API 199–200, 218
SensorBase 200, 202, 209
SensorReading 202
SensorReadingEventArgs 202, 223
service provider 110, 229
Service References 356
SetSharingMode 374, 396
SetSource 162, 316, 329
SetText 44, 239
Settings application 75, 88–89, 96, 110
settings page 278
SetView 354
ShapeInfo 382, 391
share menu 180
Share Picker 180
shared resource 165
SharedGraphicsDeviceManager 374, 397, 403
SharedPreferences 18
ShareLinkTask 96
ShareStatusTask 96
sharing pictures 180
ShellTile 49, 247
ShellToast 243
ShellToastNotificationReceived 238
ShowCamera 152
shutter button. *See* camera button
ShutterKeyHalfPressed 163
ShutterKeyPressed 163
ShutterKeyReleased 163
side loading 442
signage 7

SignalStrength 196–197
Silverlight rendering 374, 396
Silverlight Toolkit 15, 301–308
sine waveform 329
SineWaveformOscillator 330, 332
single-line editing 298
SIP. *See* software input panel
SketchFlow 429
SkipNext 194, 196
SkipPrevious 194, 196
SkyDrive 173, 180
SmallChange 301, 319
Smooth Streaming 334–340
SmoothStreamingMedia-Element 335–340
SmoothStreamingMedia-ElementState 337
SmoothStreamingSource 337
SMS text 15, 58
SmsComposeTask 96, 99
social networking 96, 110, 177, 180
SocketAsyncEventArgs 253, 255
SocketError 254
SocketType 252
soft keys 13
software input panel 5–6, 22, 41, 299
SolidColorBrush 42, 154, 354
Solution Explorer 32, 356
sound file 82
SoundEffectInstance 189, 194
SoundState 190
spacer column 207
splash screen 50
SplashScreenImage.jpg 32, 50
sprite sheet 377, 412–413, 417, 420
SpriteBatch 396–397, 403, 410, 412, 416, 419
SpriteFont 394
SQL CE. *See* Microsoft SQL Server Compact
SQLite 16, 18
StackPanel 41, 80, 95, 246, 270, 288, 354, 432
StandardTileData 49, 247
start button 37, 58, 64, 68, 73, 164, 261
start screen 49–50, 69, 111, 164, 247
StartDate 117
starting location 343
StartTime 118
StartTimeInclusive 117
startup URI 34
state dictionary 60
StateChanged 267
StaticResource 80, 94, 291, 433
status bar 37
StatusChanged 349
StatusCode 243
StatusDescription 243
StorageKind 110
storyboard editor 15
Stream 154–155, 158, 242, 252, 316
StreamInfo 339
streaming media 334
StreamResourceInfo 191, 365
street address 356
StringBuilder 350
StringFormat 63, 81, 115, 435
StrokeThickness 293, 306
student accounts 439
submission process 26, 441
SubmissionInfo 440
SubmitChanges 135, 138, 141
submitting applications 438
subtle text style 291
SupportedOrientations 36, 39, 286
SuppressFrame 389
SVG. *See* Scalable Vector Graphics
SwitchForeground 302
system theme 266
System.Data.Linq 122, 133, 135, 142
System.Data.Linq.Mapping 133
System.Device 347
System.Device.Location 11, 27, 346
System.IO 122, 129
System.IO.IsolatedStorage 122, 127, 129
System.Net 10, 17, 19, 27, 240
System.Net.NetworkInformation 231
System.Net.Sockets 10, 17, 19
System.Windows.Input 300
System.Windows.Media 17, 19, 27
System.Windows.Media.Animation 15
System.Windows.Media.Imaging 167–168
System.Windows.Shapes 38, 363
System.Windows.xaml 363
System.Xml.Serialization 130
SystemTray 37, 318

T

tab control. *See* pivot
target operating system version 32
targeted distribution 441–442
TargetName 288
TargetProperty 288
Task Switcher 58, 64, 66, 68, 153
TaskEventArgs 104, 108
TaskID 177
TaskResult 106, 109, 153, 367
Tasks API 93
TCP 228, 249
tel URL 15
TelephoneNumber 299
tesla 213
Text input scope 41
text messaging application 99
text styles 291
Text1 243
Text2 243
TextBlock 290, 292, 432
TextBox 298–299, 432
text-decoration 364
Texture 396, 403
Texture2D 396, 408, 413
TextWrapping 299
theme background 291
theme colors 266, 291
theme resource 36, 38, 40, 81, 95, 280, 293, 302
Themes.xaml 363
third party applications 108, 172
ThreadPool 232
thumbnail 163, 175
thumbstick 407–411
tile notification 233, 237, 239
tiles 8, 11, 18, 49, 65, 228, 233, 245
TimeBetweenUpdates 201, 209, 215, 218, 222
TimePicker 301, 303
TimeSpan 91, 183, 185, 209, 215, 218, 222, 392
TimeStamp 416
TimeTypeConverter 303
title layer 268
title styles 291
TitlePanel 36, 288, 431
TitleTemplate 274
Today Screen 13

ToggleButton 297, 302
ToggleSwitch 301–302
token 179, 241, 247
tombstone recovery 69
tombstoned state 60, 66–73, 104
tombstoning 66, 69–73
toolbar. *See* application bar
toolbox 262
top layer 268
torus 377
TotalSeconds 64, 68–69
TotalTime 376
Touch API 407
touch events 41
touch gestures 41–42, 415
touch manipulation events 296
touch point 416
touch target 296
touch-and-drag 417
TouchCollection 409
TouchLocation 407, 410
TouchLocationState 407, 410
TouchPanel 306, 407, 409, 415–416
TrackEnded 196
TrackInfo 337
transparency 51
transparent pixels 295
TResult 142
trial licensing 52, 101, 441
TrueHeading 214–215
TryCreate 244, 246
TryGetValue 128, 238, 273, 278
TryParse 246, 248
Twitter 96
TwoPi 386
TwoWay 434, 436
type parameter 200, 202
typography 7

U

UDP 228, 249
UIAccelerometer 16
UIElement 43, 166, 265, 307, 396
UIElementRenderer 394, 396–397, 403
UIKit 15
UINavigationController 15
UnauthorizedAccessException 28, 201
UnhandledException 90
unicast 249
UniformToFill 294
UnitX 381–382
UnitY 385
UnitZ 385
Universal Volume Control 195–197
UnloadedPivotItem 281
UnloadingPivotItem 281
update loop 412
UpdateInterval 375
UpdateSource 140
UriKind 46, 315, 361
Url input scope 42
USB port 33
User Datagram Protocol 10, 249
user defaults 16
user experience 7
User Experience Design Guidelines 35, 42, 266
user interface guidelines 15
user interface thread 95, 162, 164, 202, 223, 232
user preferences 127
user reviews 438
UserAction 195
UserControl 432
UserData API 110–119
UserExtendedProperties 27
UserIdleDetectionMode 419
user-scalable 362
UTF8 253
UVC. *See* Universal Volume Control

V

ValidateWaveFormat 331
value converters 45, 436
ValueChanged 303
VCR buttons 312
Vector2 396, 403, 408
Vector3 200, 210, 215, 218–219, 381–382, 393
velocity 212
vertical intensity 213, 216
VerticalAccuracy 346
VerticalDrag 415–416
VerticalScrollBarVisibility 298
vertices 377
VibrationController 392
VideoBrush 162, 273
VideoMediaStreamSource 324
VideoPlayer 16, 19
VideoView 19
view coordinate 381
viewfinder 161, 164–165
viewport 362
virtualization extensions 21
Visual Basic 20
visual cue 417
visual design 423
visual designer 37
visual editor 15, 262
visual feedback 414
Visual Studio 4, 7, 15, 20, 30, 203, 210, 228
Visual Studio Express 423
visual tree 263
VisualStateGroup 288
VisualStateManager 287–290
VisualTreeHelper 263
voice recording 183

W

wait time 244, 247
WAV 312, 331
wave file 187
Waveform Audio File Format 329
WaveFormatEx 188, 322, 330–331
WDDM 1.1 driver 21
Weather Channel 8
web header 241
web request 239
web service 11
web storage 358
WebBrowser 27, 297, 342, 360, 364, 366
WebBrowserTask 96, 297, 342, 359
WebCamera 16
WebClient 10, 17, 19
weekly recurrence 81
Wi-Fi 85, 228, 230, 232, 342
Win32 API 12
windows button 58
Windows CE 4
Windows Live 9, 96, 110, 174, 176
Windows Mobile 3–4, 7, 12
Windows Phone 6.5 3–4, 12
Windows Phone Application project 425
Windows Phone Audio Playback Agent 31, 194
Windows Phone Audio Streaming Agent 31
Windows Phone Class Library 31

Windows Phone Databound Application 31
Windows Phone Developer Tools 20, 262
Windows Phone Landscape Page 39
Windows Phone Marketplace 11, 22
Windows Phone Panorama Application 31, 268
Windows Phone Panorama Page 39, 268
Windows Phone Pivot Application 31, 110, 276
Windows Phone Pivot Page 39, 111, 228, 235, 276
Windows Phone Portrait Page 39
Windows Phone Scheduled Task Agent 31, 86
Windows Phone Silverlight and XNA Application 31, 373, 378
Windows Phone User Control 39, 204
Windows Presentation Foundation 13, 430, 432
Wings 3D 377
WinRT 430
wireless access point 346
wireless service provider 229
Wireless80211 231
WMA 107, 312
WMAppManifest.xml 25, 32, 86, 97, 111, 116, 201, 346
WMV 312, 314
word correction 41
workflow 15
world coordinate 381
world matrix 381
world origin 381
WPConnect Tool 24, 33, 153, 196
WPconnect.exe Tool 175
WPF. *See* Windows Presentation Foundation
WPNotification 243–244
WrapPanel 301
WriteableBitmap 153, 157, 166, 168
WriteAcquiredItem 192
WriteRecentPlay 192

X

x:Key 436
x:Name attribute 263
XAML designer 423
XAML editor 262
XAP Deployment Tool 23
.xap file 18, 82, 107, 295, 311, 359, 395
Xbox 360 12, 20, 372
Xbox Live 9, 11
X-DeviceConnectionStatus 242
X-MessageID 241
xml namespace 302
xmlns 205, 243, 269, 336, 352
XmlSerializer 130–131
XNA event system 183
XNA Framework 13, 15, 31, 172, 185, 189, 372
XNA Game Studio 4, 12, 20, 372
XNA rendering 374, 394
.xnb file 373, 377
X-NotificationClass 241, 244, 247
X-NotificationStatus 242
X-SubscriptionStatus 242
X-WindowsPhone-Target 241, 244, 247

Y

Yaw 220, 222, 225
yearly recurrence 81

Z

zip archive 18
ZoomBarVisibility 352
ZoomLevel 343–344, 352
Zune 6–7, 19–20, 153, 173–174

MOBILE TECHNOLOGY/WINDOWS

Windows Phone 7 IN ACTION

Binkley-Jones • Perga • Sync

Windows Phone 7 is a powerful mobile platform sporting the same Metro interface as Windows 8. It offers a rich environment for apps, browsing, and media. Developers code the OS and hardware using familiar .NET tools like C# and XAML. And the new Windows Store offers an app marketplace reaching millions of users.

Windows Phone 7 in Action is a hands-on guide to programming the WP7 platform. It zips through standard phone, text, and email controls and dives head-first into how to build great mobile apps. You'll master the hardware APIs, access web services, and learn to build location and push applications. Along the way, you'll see how to create the stunning visual effects that can separate your apps from the pack.

What's Inside

- Full introduction to WP7 and Metro
- HTML5 hooks for media, animation, and more
- XNA for stunning 3D graphics
- Selling apps in the Windows Store

Written for developers familiar with .NET and Visual Studio. No WP7 or mobile experience is required.

Timothy Binkley-Jones is a software engineer with extensive experience developing commercial IT, web, and mobile applications. **Massimo Perga** is a software engineer at Microsoft and **Michael Sync** is a solution architect for Silverlight and WP7.

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/WindowsPhone7inAction

Free eBook

SEE INSERT

> “Definitely recommended!”
>
> —Vipul Patel, Amazon.com

> “Top resource for Windows Phone developers.”
>
> —Loïc Simon, Solent SAS

> “A great handbook for climbing the WP7 ladder.”
>
> —Francesco Goggi
> Magneti Marelli

> “Gives you a kickstart in Windows Phone development.”
>
> —Mark Monster
> Monster Consultancy

ISBN 13: 978-1-617290-09-1
ISBN 10: 1-617290-09-2

53999

9 781617 290091

MANNING

$39.99 / Can $41.99 [INCLUDING eBOOK]